MATLAB AND OCTAVE PROGRAMMING FOR STEM APPLICATIONS (SMITH)

Cary Smith
Oxnard College

CHAPTER OVERVIEW

Front Matter

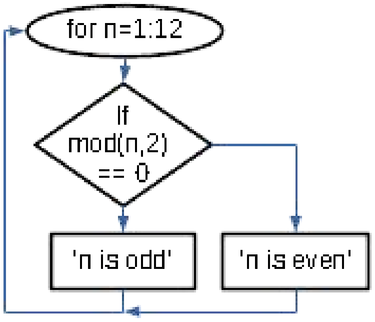
MATLAB/Octave Programming for STEM Applications

Carey A. Smith

Oxnard College, Ventura County Community College District, CA

CareySmith@vcccd.edu

Version: 2024 June 4. The chapter on interpolation and curve fitting was substantially rewritten and moved to become chapter 10, so that it follows the random numbers chapter. So, the previous chapters 12 to 15 were renumbered.



July 28, 2022

InfoPage

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the hundreds of other texts available within this powerful platform, it is freely available for reading, printing and "consuming." Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects.

Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts’ web based origins allow powerful integration of advanced features and new technologies to support learning.

TABLE OF CONTENTS

Front Matter

- TitlePage
- InfoPage
- Table of Contents
- Licensing

Licensing

Preface

- 1.1: Notes to the Instructor

Authors and Sources

Purpose of This Book

How this Book Was Assembled

Back Matter

- Index
- Glossary
- Detailed Licensing

1: Modeling, Simulation, and MATLAB’s Interpreter

- 1.1: Obtaining MATLAB and Octave
- 1.2: Modeling
- 1.3: A Glorified Calculator
    - 1.3.1: log(), round(), floor(), and ceil() functions
    - 1.3.2: Trigonometry Functions
- 1.4: MATLAB User Interface
- 1.5: Octave User Interface
- 1.6: Variables and Assignment Statements
    - 1.6.1: More About Valid Variable Names
- 1.7: Clearing the Workspace, Readability Tips
- 1.8: Errors
- 1.9: Built-in Function Documentation
- 1.10: Chapter Review
- 1.11: Chapter 1 videos English and Espanol
- 1.12: Exercises

2: MATLAB/Octave Scripts

- 2.1: How to Create a Script File and Valid File Names
- 2.2: Your First Script
    - 2.2.1: .mlx Live Editor Files
- 2.3: Examples of assignment scripts

- 2.4: Why Scripts?
- 2.5: The Fibonacci Sequence
- 2.6: Floating-Point Numbers
- 2.7: Scientific Notation for Large and Small Numbers
- 2.8: Number formats
- 2.9: Complex Numbers
- 2.10: Script Comments
- 2.11: Script Documentation
- 2.12: Assignment and Equality
- 2.13: Understanding Editor Warning and Error messages
- 2.14: Debugging Directory Errors
- 2.15: Chapter Review
- 2.16: Chapter 2 Scripts videos English and Espanol
- 2.17: Exercises
  - 2.17.1: More Exercises

3: for Loops, Ploting Points, Sequences, and Series

- 3.1: Updating Variables
- 3.2: Bug Taxonomy
- 3.3: Absolute and Relative Error
- 3.4: for Loops
  - 3.4.1: for Loop Supplemental Material
- 3.5: Plotting Points
  - 3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop
- 3.6: Sequences
  - 3.6.1: Sequences Additional Material
- 3.7: Series
  - 3.7.1: Series Additional Material
- 3.8: Generalization
- 3.9: Incremental Development
- 3.10: for Loop Incremental Development Exercise
- 3.11: Chapter Review
- 3.12: Chapter 3 for loop Videos in English and Espanol
- 3.13: Exercises

4: Vectors

- 4.1: Creating Vectors
- 4.2: Methods to Create Vectors in MATLAB
- 4.3: Vector Arithmetic
- 4.4: Selecting Elements
- 4.5: Indexing Errors
- 4.6: Vectors and Sequences
- 4.7: disp() and num2str() Functions and Concatenation
- 4.8: Plotting Vectors
  - 4.8.1: Plotting Vectors Additional Material
- 4.9: Recursive for Loops
- 4.10: Common Vector Operations

- 4.11: Common Vector Functions
- 4.12: Chapter Review
- 4.13: Chapter 4 Vector Videos In English and Espanol
- 4.14: Exercises

5: User-Defined Functions

- 5.1: Name Collisions
- 5.2: Defining Functions
  - 5.2.1: Function Case Study (Lambert)
- 5.3: Function Documentation
- 5.4: Naming Functions
  - 5.4.1: Function Names and MATLAB Keywords
- 5.5: Functions With Multiple Input Variables
  - 5.5.1: Multiple Inputs Examples and Exercises
  - 5.5.2: Mutiple Inputs Example- Ideal Gas law (Lambert)
- 5.6: Functions with Multiple Outputs
- 5.7: Matlab Functions with No Inputs
- 5.8: Matlab Functions with No Outputs
- 5.9: Matlab Functions with No Inputs and No Outputs
- 5.10: Functions with variable numbers of inputs or outputs
- 5.11: How Functions Work
- 5.12: Chapter Review
- 5.13: Chapter 5 Function videos English and Espanol
- 5.14: Exercise

6: Conditionals Part 1

- 6.1: Relational Operators
  - 6.1.1: Relational Operators List
  - 6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction
- 6.2: if Statement
- 6.3: if, else, elseif
- 6.4: Incremental Development
- 6.5: Logical Functions
- 6.6: Nested for Loops
- 6.7: Putting It Together
- 6.8: Encapsulation and Generalization
- 6.9: Adding a continue Statement
- 6.10: Chapter Review
- 6.11: Chapter 6 if-logic videos English and Espanol
- 6.12: Exercise

7: Conditionals Part 2- Series, Switch Logic, Error Checking, and While Loops

- 7.1: if, elseif, else with Error Checking
- 7.2: Flowcharts
- 7.3: The input() Function
- 7.4: Compound Logical Operators

    - 7.4.1: Compound if, elseif Logic Examples
- 7.5: Working with Conditions
- 7.6: Shortcut and, or, Logical Operations
- 7.7: for loops with break Logic
- 7.8: 'while' Loops
- 7.9: 'switch' Logic
- 7.10: The find() function
- 7.11: Chapter 7 Conditionals, Part 2, Videos English and Espanol

8: User-defined Functions of Vectors

- 8.1: Functions and Vectors
    - 8.1.1: Built-in MATLAB Functions of Vectors
    - 8.1.2: User-Defined Functions of Vectors Examples and Exercises
- 8.2: Computing with Vectors
- 8.3: Debugging in Four Acts
    - 8.3.1: Debugging Theorems all in 1 Place
- 8.4: Chapter Review
- 8.5: Chapter 8 Function Videos English and Espanol

9: MATLAB Random Number and Statistics Functions

- 9.1: MATLAB Random Number Functions
- 9.2: MATLAB Statistics Functions
- 9.3: Statistical Plots
- 9.4: Random Number Applications
- 9.5: Chapter 9 Random Number Videos English and Espanol

10: Interpolation and Curve Fitting

- 10.1: Linear Interpolation
    - 10.1.1: Linear Interpolation Problem Set
- 10.2: Nonlinear Interpolation
- 10.3: Linear and Polynomial Curve Data Fitting
- 10.4: Polynomial Fitting Problem Set
- 10.5: Interactive Fitting Tools
- 10.6: Chapter 10 Curve Fitting Videos in English and Espanol

11: Zero Finding and Fourier Transforms

- 11.1: Solving Nonlinear Equations with fzero()
- 11.2: fzero() Examples and Exercises
- 11.3: Using fzero() to find the intersection of 2 functions
- 11.4: Subfunctions
- 11.5: Anonymous (or In-Line) Functions
- 11.6: Debugging- More Name Collisions
- 11.7: Fourier Transforms
- 11.8: Chapter Review
- 11.9: Chapter 11 Zero finding Videos English and Espanol
- 11.10: Exercises

12: MATLAB Matrices (Arrays)

- 12.1: Creating Matrices and Arrays
- 12.3: Common Matrix Functions
- 12.4: Matrix Indexing
- 12.5: Matrix Operations, Introduction
- 12.6: Matrix Tables and Nested Loops
- 12.7: Dot Product
- 12.8: Cross Product and Torque
- 12.9: Chapter 12 Matrix Videos English and Espanol

13: Linear Algebra

- 13.1: Basic Matrix Multiplication
- 13.2: Matrix Multiplication Applications
- 13.3: Matrix Inverse, Rank and Determinant
- 13.4: Solving Systems of Equations
- 13.5: Systems of Equations Examples and Exercises
- 13.6: Over-Determined Systems of Equations
- 13.7: Chapter 13 Linear Algebra Videos in English and Espanol

14: Advanced Plotting

- 14.1: scatter, scatter3
- 14.2: Plot 3D Surface and Contour Plots
- 14.3: Advanced Plot Formatting
- 14.4: Log plots
- 14.5: Subplots
- 14.6: Chapter 14 Advanced Plotting Videos in English and Espanol

15: Formatted Input and Output

- 15.1: sprintf() and fprintf(), Introduction
- 15.2: fprintf() to a file
- 15.3: Reading from an Excel file in MATLAB and Octave
- 15.4: Chapter 15 Videos English and Espanol

16: MATLAB Structures and Data Types

- 16.1: Numeric data types
- 16.2: Comparing Character Strings
- 16.3: table Data Type
- 16.4: struct Data Type
  - 16.4.1: struct Examples and Exercises
- 16.5: cell array
- 16.6: Chapter 16 Data Types Videos in English and Espanol

17: Top-Down Design, Agile Software Development, and Object-Oriented Programming in MATLAB

- 17.1: Top-Down Design
- 17.2: Agile Software Development

- 17.3: Object-Oriented Programming, Part 1
- 17.4: Object-Oriented Programming, Part 2
- 17.5: Chapter 17 Videos in English and Espanol

18: Integration and Differentiation

- 18.1: Integration- Computing the Area Under a Curve
    - 18.1.1: Problem Set
    - 18.1.2: The integral() function
- 18.2: Numerical Differentiation
- 18.3: Chapter 18 Differentiation and Integration Videos in English and Espanol

19: Ordinary Differential Equations

- 19.1: Functions and Files
- 19.2: Differential Equations
- 19.3: Euler’s Method
- 19.4: Implementing Euler’s Method
- 19.5: Solving ODEs with ode45
    - 19.5.1: ODE45 Examples and Exercises
- 19.6: Time Dependence
- 19.7: What Could Go Wrong?
- 19.8: Labeling Axes
- 19.9: Chapter Review
- 19.10: Chapter 19 ODE Videos in English and Espanol
- 19.11: Exercise

20: Symbolic Processing

Detailed Licensing

MATLAB Functions, Commands and Special Characters

Index

Glossary

Detailed Licensing

Licensing

This OER textbook is covered by a creative commons license. Most sections are either CC BY-NC or CC BY-NC-SA.

A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.

Preface

MATLAB is a widely used analysis tool by scientists and engineers. MATLAB has built-in graphing capabilities and many analysis functions, including matrix multiplication and matrix inverse. It can be run interactively, which greatly helps checking & debugging. This text emphasizes practical applications in engineering, physics, mathematics and biology.

OCTAVE is a free program that runs almost all MATLAB code in this course.

There are a few differences between OCTAVE and MATLAB in implementation. These differences are described where needed.

Watch these MATLAB overview videos:

https://www.mathworks.com/videos/matlab-overview-61923.html

https://www.youtube.com/watch?v=OHxR8iMHDWw

This video shows how Penn State students use MATLAB in internships:

https://www.mathworks.com/videos/penn-state-students-transition-to-industry-78737.html

Authors and Sources

Carey A. Smith, Editor. Oxnard College, California. CareySmith@vcccd.edu

I have taught Programming and Problem-Solving in MATLAB since 2016. I spent 30+ years in industry as a software engineer, using MATLAB for 15+ years.

This LibreTexts book includes contributions from OER textbooks and materials from the following authors, as well my own materials. I am very grateful to each of these authors for their contributions.

Allen B. Downey, Physical Modeling in MATLAB, version 4.0, 2021,
Creative Commons Attribution-NonCommercial 4.0 Unported License
https://greenteapress.com/wp/physical-modeling-in-matlab/

https://www.oercommons.org/courses/physical-modeling-in-matlab
downey@allendowney.com

Serhat Beyenir, A Brief Introduction to Engineering Computation with MATLAB, 2020. Licensed under CC BY.
https://eng.libretexts.org/Bookshelv...TLAB_(Beyenir)

Serhat_Beyenir@bcit.ca, British Columbia Institute of Technology

Adam L. Lambert, Zero to MATLAB, 2020, License: Creative Commons Attribution Non-Commercial
https://www.oercommons.org/courses/zero-to-matlab

adam.lambert@oregonstate.edu

Troy Siemers, An Introduction to MATLAB and MATHCAD, 2014, License: Creative Commons Attribution Non-Commercial
https://www.oercommons.org/courses/a...ab-and-mathcad
https://eng.libretexts.org/Bookshelv...hcad_(Siemers)
siemerstj@vmi.edu

Donald Dabdub, http://albeniz.eng.uci.edu/mae10/PDF...lowControl.pdf and Relational_Logical_Operators.pdf
http://albeniz.eng.uci.edu/dabdub, University of California, Irvine

Brian Vick, MatlabCommands_Brian_Vick.pdf

bvick@vt.edu, Virginia Tech

Purpose of This Book

The purpose of this book is to provide a comprehensive textbook for California's C-ID ENGR 220 Programming and Problem-Solving in MATLAB.
Such a course should cover the following topics:

Overview of computer systems and the MATLAB environment
Variables, expressions, and order of operation
Elementary functions
Array definitions and operations
Computational problem-solving methodology
Pseudocode, flowcharts, and documentation
Formatted input and output
Data Files
Plotting
Selection programming structures
Repetition programming structures
MATLAB Functions and user-defined functions
Recursion
Data structures
Sorting and searching
Object-oriented programming
Numerical Analysis Techniques (embedded within topics above):
Solving systems of linear equations
Vector analysis
Data Interpolation
Least-squares regression
Numerical Differentiation and Integration
Solving ordinary differential equations
Series approximation and error
Solving equations of one variable

Since no single Open Educational Resource (OER) textbook covered all these topics, I assembled parts of other books and added my own materials. This book is intended to cover these topics and provide sufficient examples and exercises to be an effective learning tool.

If you find errors or have suggestions for improvements, send them to me at CareySmith@vcccd.edu.

How this Book Was Assembled

The books listed in the Authors and Sources section have some very good materials. For the 2021-2022 school year, I used those sources plus my own materials to teach my MATLAB programming course. In order to make it more convenient for both the students and the teacher, I have assembled these materials into a single textbook. I felt that many chapters could benefit from additional examples and exercises, which I added. Additional topics were also added to many chapters.

The following outlines how each source was used in assembling this textbook.

1: Modeling, Simulation, and MATLAB’s Interpreter	Downey and Smith
2. Scripts	Downey and Smith
3: for Loops and Basic 2D Plots	Downey and Smith
4: Vectors	Downey and Smith
5: User-Defined Functions	Downey, Lambert, and Smith
6: Conditionals, part 1	Downey, Dabdub, and Smith
7: Conditionals with Series, Switch Logic, and While Loops	Smith and Siemers
8: User-Defined Functions of Vectors	Downey and Smith
9: MATLAB Random Number and Statistics Functions	Smith
10:. Zero Finding and Fourier Transforms	Downey and Smith
11: MATLAB Matrices	Siemers and Smith
12: Linear Algebra	Smith and Siemers
13: Formatted Input, Output	Beyenir and Smith
14 Matlab Structures and Data Types	Lambert and Smith
15: Advanced Plotting	Lambert and Smith
16: Interpolation and Curve Fitting	Beyenir, Siemers, and Smith
17: Top-down Design, Agile Software Development, and Object-Oriented Programming in MATLAB	Smith
18: Integration, Differentiation	Beyenir and Smith
19: Ordinary Differential Equations	Downey and Smith
20. Symbolic Processing	Smith

CHAPTER OVERVIEW

Back Matter

CHAPTER OVERVIEW

1: Modeling, Simulation, and MATLAB’s Interpreter

1.1: Obtaining MATLAB and Octave
1.2: Modeling
1.3: A Glorified Calculator
  1.3.1: log(), round(), floor(), and ceil() functions
  1.3.2: Trigonometry Functions
1.4: MATLAB User Interface
1.5: Octave User Interface
1.6: Variables and Assignment Statements
  1.6.1: More About Valid Variable Names
1.7: Clearing the Workspace, Readability Tips
1.8: Errors
1.9: Built-in Function Documentation
1.10: Chapter Review
1.11: Chapter 1 videos English and Espanol
1.12: Exercises

The procedures to to obtain and install either MATLAB or Octave are in this section of "Front Matter":

Obtaining_MATLAB_and_Octave

Note from Carey Smith: The majority of the first chapters are from Allen Downey's OER book. (Details are in "Front Matter".) These first chapters introduce the basic skills needed to program and solve all sorts of problems in MATLAB. Later chapters applies these skills to modeling physical systems. This book covers many topics in addition to modeling physical systems. At the end of every chapter are videos if you want more help in understanding the material.

Note from Allen Downey: Before we can build any models of physical systems, it’ll help to have a high-level understanding of what a model is. We’ll also need to familiarize ourselves with the tools we use to build them. In this chapter, we’ll look at the modeling process and introduce MATLAB, the programming language we’ll use to represent models and run . At the end of the chapter you’ll find exercises you can use to test your knowledge.

.

This page titled 1: Modeling, Simulation, and MATLAB’s Interpreter is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

1.1: Obtaining MATLAB and Octave

You need to install either MATLAB or Octave for this course. MATLAB has some advantages:

- MATLAB has better error checking and better help. MATLAB's documentation is better, but you can most of MATLAB's documentation without purchasing a license.
- MATLAB runs faster. This is not an issue for this class' assignments, but is an issue for large analysis code you might use professionally.

1.1.1: Obtaining MATLAB

MATLAB can be installed on either Windows or Mac computers. The basic MATLAB student license is $49, which is a great deal. (For more features see the "The MATLAB & Simulink Student Suite" section below.) The MATLAB licenses are valid for as long as you are a student anywhere. The link to purchase the MATLAB license is:

https://www.mathworks.com/store/link/products/student/new?s_iid=htb_buy_gtwy_cta3

The links to install it on a Windows or Apple computer are there.

MATLAB also has a free on-line version which is adequate for this course. The online version has limitations that only apply to professional applications, such as integrating it with hardware. A description of the online version's limitations is here:

https://www.mathworks.com/products/matlab-online/limitations.html

If you have a Chromebook, consider using Octave. MATLAB can be installed on some (but not all) Chromebooks. The process can be difficult because Chromebooks use the Linux operating system. These instructions may or may not work for you:

https://www.linuxfordevices.com/tutorials/linux/installing-matlab-on-linux

Another option for a Chromebook is to use the MATLAB on-line version.

1.1.2: The MATLAB & Simulink Student Suite

If you wish, you can choose the MATLAB & Simulink Student Suite is $99. This includes a number of toolboxes and Simulink, which you may find useful in future classes. This is also a great deal, but is not needed for this class.

Simulink is an advanced software program that let's you model systems with feed back in a graphical way. You can create "block diagrams", as described in these videos. It will not be used in this class, but it is likely to be useful for you is your major is mechatronics, electrical engineering, or computer engineering. This video gives a quick overview.

hat Is Simulink? | Simulink Overvie
LAB
nk?
Watch on YouTube

1.1.3: Obtaining Octave

Octave can be downloaded at this link:

https://www.gnu.org/software/octave

Octave for a MAC is here:

https://ports.macports.org/port/octave/

Octave Online runs online without downloading at this link. You need to create an account.

Octave Online · Cloud IDE compatible with MATLAB (octave-online.net)

Octave Online also runs on Chrome.

Octave Online - Chrome Web Store (google.com)

This video shows how to download and install Octave, as well as how to start using Octave:

tave Tutorial #1: Getting Started f
STEM EDU TV
Tutorial #1
ting Started
for Absolute Beginners
>> 1 + 2 = ?
Watch on YouTube

.

This page titled 1.1: Obtaining MATLAB and Octave is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

1.2: Modeling

When I say “modeling,” I’m talking about something like Figure 1.1. In the lower-left corner of the figure is the *system*, something in the real world we’re interested in. Often, it’s something complicated, so we have to decide which details can be left out; removing details is called *abstraction*.

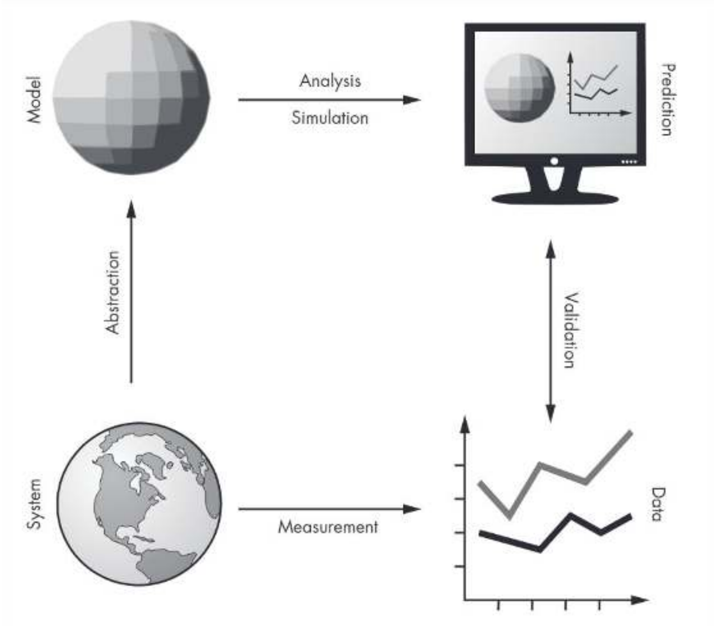


Figure 1.1: The modeling process

The result of abstraction is a *model*, shown in the upper left; a model is a description of the system that includes only the features we think are essential. A model can be represented in the form of diagrams and equations, which can be used for mathematical analysis. It can also be implemented in the form of a computer program, which can run simulations.

The result of analysis and simulation might be a prediction about what the system will do, an explanation of why it behaves the way it does, or a specific design engineered to satisfy a requirement or optimize performance.

We can validate predictions and test designs by taking measurements from the real world and comparing the data we get with the results from analysis and simulation.

For any physical system there are many possible models, each one including and excluding different features or including different levels of detail. The goal of the modeling process is to find the model best suited to its purpose (prediction, explanation, or design).

Sometimes the best model is the most detailed. If we include more features, the model is more realistic, and we expect its predictions to be more accurate. But often a simpler model is better. If we include only the essential features and leave out the rest, we get models that are easier to work with, and the explanations they provide can be clearer and more compelling.

As an example, suppose someone asked you why the orbit of the Earth is nearly elliptical. If you model the Earth and Sun as point masses (ignoring their actual size), compute the gravitational force between them using Newton’s law of universal gravitation, and compute the resulting orbit using Newton’s laws of motion, you can show that the result is an ellipse.

Of course, the actual orbit of Earth is not a perfect ellipse, because of the gravitational forces of the Moon, Jupiter, and other objects in the solar system, and because Newton’s laws of motion are only approximately true (they don’t take into account relativistic effects).

But adding these features to the model would not improve the explanation; more detail would only be a distraction from the fundamental cause. However, if the goal is to predict the position of the Earth with great precision, including more details might be necessary.

Choosing the best model depends on what the model is for. It is usually a good idea to start with a simple model, even if it’s likely to be too simple, and test whether it’s good enough for its purpose. Then you can add features gradually, starting with the ones you expect to be most essential. This process is called *iterative modeling*.

Comparing the results of successive models provides a form of *internal validation* so you can catch conceptual, mathematical, and software errors. And by adding and removing features, you can tell which ones have the biggest effect on the results, and which can be ignored. Comparing results with data from the real world provides *external validation*, which is generally the strongest test.

Figure 1.1 shows that models can be used for both analysis and simulation; in this book we will do some analysis, but the focus is on simulation. And the tool we will use to build simulations is MATLAB. So let’s get started.

This page titled 1.2: Modeling is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **1.1: Modeling** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

1.3: A Glorified Calculator

MATLAB is a programming language with features that support modeling and simulation. It has a lot of features, so it can seem overwhelming, but at heart MATLAB is a glorified calculator. When you start MATLAB you should see a window entitled MATLAB that contains smaller windows entitled Current Folder, Command Window, and Workspace. In Octave, Current Folder is called File Browser.

1.3.0.1: The Interpreter

The Command Window runs the *interpreter*, which allows you to enter *commands*; once entered, the interpreter executes the command and prints the result. Initially, the Command Window contains a welcome message with information about the version of the software you're running, followed by a *prompt*, which looks like this:

```
>>
```

The `>>` symbol prompts you to enter a command. The simplest kind of command is a mathematical *expression*, like `2 + 1`. If you type an expression and then press (or), MATLAB *evaluates* the expression and prints the result.

```
>> 2 + 1
ans = 3
```

Just to be clear: in this example, MATLAB displayed `>>`; I typed `2 + 1` and then hit , and MATLAB displayed `ans = 3`.

In this expression, the plus sign is an *operator* and the numbers `2` and `1` are *operands*. An expression can contain any number of operators and operands. You don't have to put spaces between them; some people do and some people don't. Here's an example with no spaces:

```
>> 1+2+3+4+5+6+7+8+9
ans = 45
```

Speaking of spaces, you might have noticed that MATLAB puts a blank line between `ans =` and the result. In my examples I'll leave it out to save room.

The other arithmetic operators are pretty much what you would expect. Subtraction is denoted by a minus sign (`-`), multiplication is designated by an asterisk (`*`), division is denoted by a forward slash (`/`).

```
>> 2*3 - 4/5
ans = 5.2000
```

Another common operator is exponentiation, which uses the `^` symbol, sometimes called "caret" or "hat." So, 2 raised to the 16th power is

```
>> 2^16
ans = 65536
```

The order of operations is what you would expect from basic algebra: exponentiation happens before multiplication and division, and multiplication and division happen before addition and subtraction. If you want to override the order of operations, you can use parentheses.

```
>> 2 * (3-4) / 5
ans = -0.4000
```

When I added the parentheses, I removed some spaces to make the grouping of operands clearer to a human reader. This is the first of many style guidelines I will recommend for making your programs easier to read. Style doesn't change what the program does; the MATLAB interpreter doesn't check for style. But human readers do, and the most important human who will read your code is you.

And that brings us to the First Theorem of Debugging:

> *Readable code is debuggable code.*

It's worth spending time to make your code pretty; it will save you time !

1.3.0.1: Math Functions

MATLAB knows how to compute pretty much every math function you've heard of. For example, it knows all the trigonometric functions—here's how you use them:

```
>> sin(1)
ans = 0.8415
```

This command is an example of a *function call*. The name of the function is `sin`, which is the usual abbreviation for the trigonometric sine. The value in parentheses is called the *argument*.

The trig functions `sin`, `cos`, and `tan` —among many others—work in radians, so the argument in the example is interpreted as 1 radian. MATLAB also provides trig functions that work in degrees: `sind`, `cosd`, and `tand`.

Some functions take more than one argument, in which case the arguments are separated by commas. For example, `atan2` computes the inverse tangent, which is the angle in radians between the positive x-axis and the point with the given x- and y-coordinates.

```
>> atan2(1,1)
ans = 0.7854
```

If that bit of trigonometry isn't familiar to you, don't worry about it. It's just an example of a function with multiple arguments.

MATLAB also provides exponential functions, like `exp`, which computes e raised to the given power. So `exp(1)` is just e:

```
>> exp(1)
ans = 2.7183
```

The inverse of `exp` is `log`, which computes the logarithm base e:

```
>> log(exp(3))
ans = 3
```

This example also demonstrates that function calls can be *nested*; that is, you can use the result from one function as an argument for another.

More generally, you can use a function call as an operand in an .

```
>> sqrt(sin(0.5)^2 + cos(0.5)^2)
ans = 1
```

As you probably guessed, `sqrt` computes the square root.

There are lots of other math functions, but this isn't meant to be a reference manual. To learn about other functions, you should read the documentation.

This page titled 1.3: A Glorified Calculator is shared under a not declared license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **1.2: A Glorified Calculator** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

1.3.1: log(), round(), floor(), and ceil() functions

You need to know some details about these MATLAB functions.

1.3.1.1: log() vs. log10() vs. ln

In algebra, we use ln(x) for the natural logarithm of x, which is base e, and we use log(x) for the logarithm base 10 of x.

For a refresher, see this Wikipedia article: https://en.wikipedia.org/wiki/Logarithm [en.wikipedia.org]

But, very early in computer programming's history, the log(x) function was defined to be the natural logarithm of x, which is base e, and so log10(x) was defined to be the logarithm base 10 of x. This tradition has been continued into current computer programming languages, possibly for compatibility. We can't change it, so we just have to remember to use log(x) for the natural logarithm of x, and log10(x) for the logarithm base 10 of x. Examples:

```
%% log base 10
c2 = log10(2)
% c10 = 0.3010
e = 2.7183
ce = log10(e)
% ce = 0.4343
c10 = log10(10)
% c10 = 1

%% Natural log of x. In math, this is ln(x)
d2 = log(2)
% d1 = 0.69315
e = 2.7183
de = log(e)
% de = 1
d10 = log(10)
%d10 = 2.3026
```

1.3.1.2: round()

Octave has a basic round() function. MATLAB has an option to round to a specified number of decimal places.

1.3.1.2.1: round() in Octave

```
y = round (x) ¶
```

Octave's round(x) function rounds the input to the nearest integer. The input may be a vector.

```
Examples:

>> round(2.3)
ans =
     2
>> round(2.5)
ans =
     3

>> round(pi)
ans = 3
```

```
>> round([2.9, 3.1, 5.4])
ans =

   3   3   5
```

Rounding for negative numbers:

```
z1 = - 34.56

>> round(z1)
ans = -35
```

Octave's round() function has no options. It accepts only 1 input, which can be a single number, a vector, or a matrix.

1.3.1.2.2: How to round to a specific number of decimal places in Octave

In Octave, you can't specify the number of decimal places directly, but this is a work-around:

Let x be the number that you want to round and n be the number of decimal places. Then use this code:

```
  y = round(x*10^n)/10^n
```

This example shows how you can round pi to 3 decimal places:

✓ Example 1.3.1.1 Octave: round pi to 3 decimal places

```
y = round(pi*10^3)/10^3
```

Solution

Add example text here.

1.3.1.2.3: round() in MATLAB

MATLAB's round(x) function rounds the input to the nearest integer when used with a single input variable. The input may be a vector or a matrix.

```
Examples:

a = 135.79246


>> round(x)
ans =
     136

b = 79.421

>> round(b) = 79
ans =
     3

>> round(pi)
ans =
     3

>> round([2.9, 3.1, 5.4])

ans =
     3     3     5
```

Rounding for negative numbers:

```
>> y = -314.159
y =
 -314.1590
>> round(y)
ans =
  -314
```

MATLAB's round() function has an option. You may round to a specified number of decimal places by adding a 2nd input.

This example shows how to round pi to 4 decimal places:

```
>> round(pi,4)
ans =
    3.1416
```

Try these examples:

```
% Round -1/3 to 2 decimal places.

x = round(-1/3, 2)

% Round 2/3 to 2 decimal places.

x = round(2/3, 2)
```

1.3.1.3: floor() function

The floor() function always rounds a number down to the next lowest integer. It rounds negative numbers towards -infinity.

It works the same in both MATLAB and Octave.

Try these examples:

```
x = floor(-1/3)
x = floor(2/3)
y = floor(4.6667)
```

1.3.1.4: `ceil() function`

MATLAB's ceil (ceiling) function always rounds up to the higher integer (towards +infinity). Try these examples:

```
x = ceil(-1/3)
y = ceil(2/3)
z = ceil(4.6667)
```

This page titled 1.3.1: log(), round(), floor(), and ceil() functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

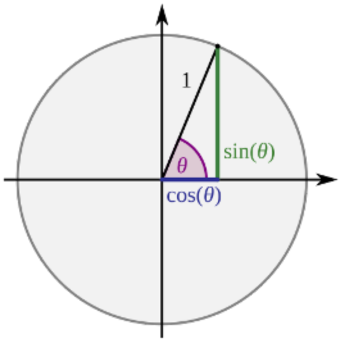
1.3.2: Trigonometry Functions

1.3.2.1: Trigonometric Relations Review

This diagram shows a right triangle with an hypotenuse of length 1. For an angle theta:

sine(theta) = x

cosine(theta) = y



https://commons.wikimedia.org/wiki/F...itskreis_1.svg. Author: Stephan Kulla (User:Stephan Kulla)

This file is made available under the Creative Commons CC0 1.0 Universal Public Domain Dedication

For a general triangle with hypotenuse, h (or circle with radius h) the relations are:

sine(theta) = x/h

cosine(theta) = y/h

tangent(theta) = y/x

The following are the corresponding MATLAB/Octave functions:

1.3.2.2: sin(), sind()

sin(theta) for theta in radians

sind(angle) for angle in degrees

1.3.2.3: cos(), cosd()

cos(theta) for theta in radians

cosd(angle) for angle in degrees

1.3.2.4: tan(), tand()

tan(theta) for theta in radians

tand(angle) for angle in degrees

1.3.2.5: atan(), atand()

atan(a) is the arctangent function, which is also called the inverse tangent function. atan(a) returns the angle in radians, with values in the interval [-pi/2, pi/2].

atand(a) is the arctangent function, which returns the angle in degrees with values in the interval [-90, 90].

1.3.2.6: atan2(y, x), atan2d(y, x)

atan2(y,x) allows the user to specify both the x & y (the cos and sin parts) of the tangent ratio. When x & y are known, atand is generally preferred. The 2 input version is advantageous because there is an ambiguity when only the tangent value is entered. This is because tangent is periodic with a period of pi radians (180 degrees).

In other words, because tan(theta) = sin(theta) / cos(theta) also = -sin(theta) / -cos(theta)

this occurs when theata2 = theta1 + pi

For example,

`tan(pi/3)` = 1.7321

and

`tan(-2*pi/3)` = 1.7321

In MATLAB and Octave, `atan(1.7321)` = 1.0472 = pi/3

The following pairs of angles illustrate that they have the same value for the tangent:

theta1 = pi/6 (30 degrees) and theta2 = -5*pi/6 (-150 degrees or +210 degrees):

tan(theta1) = sin(theta1) / cos(theta1) = 0.5000 / 0.8660 = 0.5774

tan(theta2) = sin(theta2) / cos(theta2) = -0.5000 / -0.8660 = 0.5774

theta1 = -pi/6 (-30 degrees) and theta2 = 5*pi/6 (150 degrees):

tan(theta1) = sin(theta1) / cos(theta1) = -0.5000 / 0.8660 = -0.5774

tan(theta2) = sin(theta2) / cos(theta2) = 0.5000 / -0.8660 = -0.5774

`atan2()` and `atan2d()` solve this problem because both the numerator and denominator of the tangent are inputs. Examples:

`sin(4*pi/3)` = -0.8660

`cos(4*pi/3)` = -0.5000

`atan2(-0.8660, -0.5000)` = -2.0944 = -2*pi/3

Note that the inputs for the sin() cos() and tan() functions are in radians and the outputs of the atan() and atan2() functions are in radians.

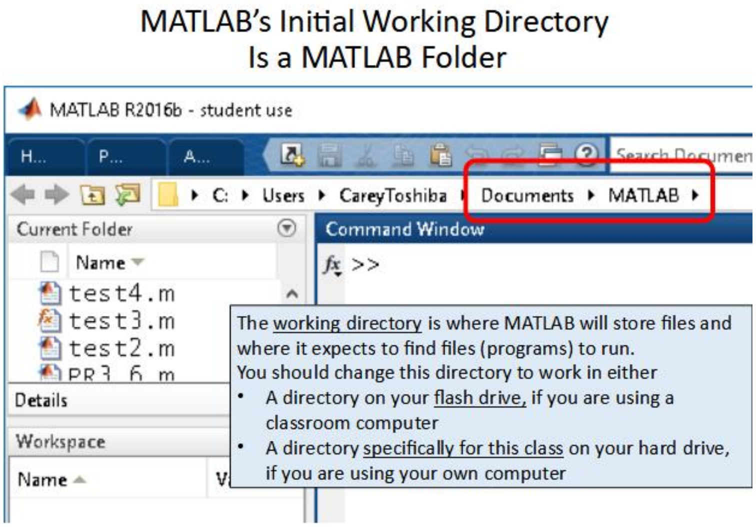
When you want to work in degrees, use atand() and atan2d().

.

1.3.2: Trigonometry Functions is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by LibreTexts.

1.4: MATLAB User Interface

By Carey A. Smith


MATLAB's Initial Working Directory Is a MATLAB Folder

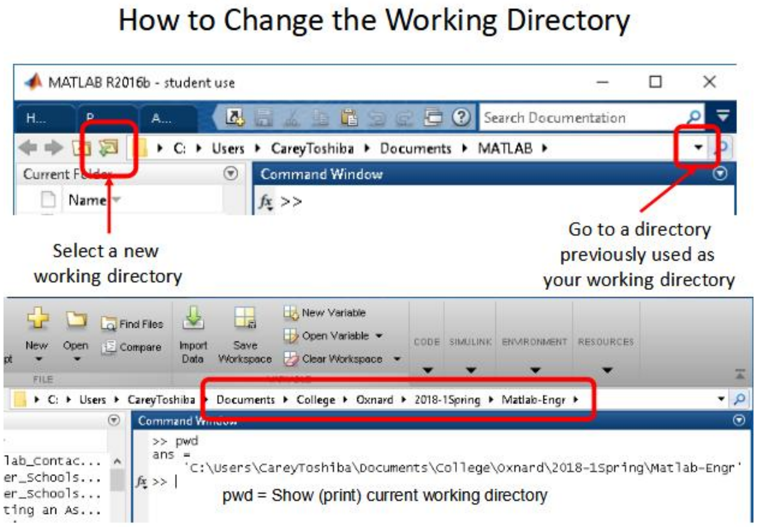
The working directory is where MATLAB will store files and where it expects to find files (programs) to run.
You should change this directory to work in either

- A directory on your flash drive, if you are using a classroom computer
- A directory specifically for this class on your hard drive, if you are using your own computer

MATLAB R2016b - student use
H...
P...
A...
Search Documen
C:
Users
CareyToshiba
Documents
MATLAB
Current Folder
Command Window
Name
test4.m
test3.m
test2.m
PR3_6.m
fx >>
Details
Workspace
Name

```
Figure \(\PageIndex{i}\): MATLAB User Interface's Initial Working Directory
```

You should keep your files in a directory that is different from MATLAB's default directory. This figure shows you how to change MATLAB's working directory to be your directory.


How to Change the Working Directory

Select a new working directory

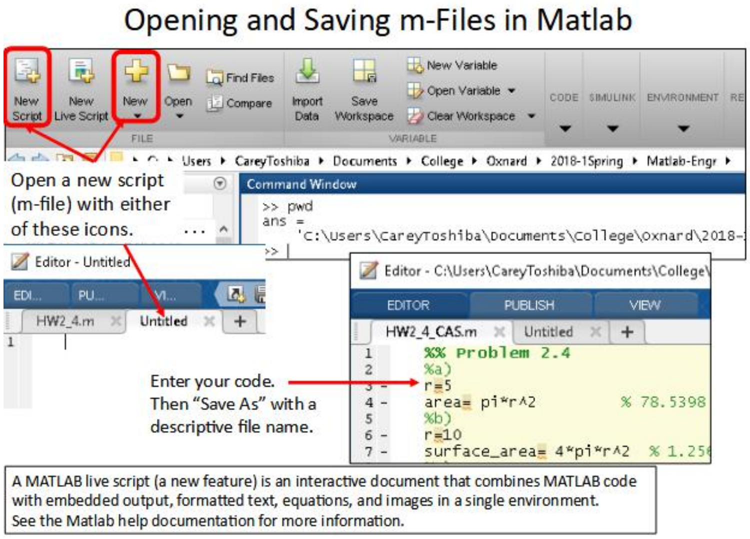
Go to a directory previously used as your working directory

```
>> pwd
ans =
    'C:\Users\CareyToshiba\Documents\College\Oxnard\2018-1Spring\Matlab-Engr'
fx >> |
```

pwd = Show (print) current working directory

MATLAB R2016b - student use
H...
P...
A...
Search Documentation
C:
Users
CareyToshiba
Documents
MATLAB
Current Folder
Command Window
Name
fx >>
New
Open
Find Files
Compare
Import Data
Save Workspace
New Variable
Open Variable
Clear Workspace
CODE
SIMULINK
ENVIRONMENT
RESOURCES
FILE
C:
Users
CareyToshiba
Documents
College
Oxnard
2018-1Spring
Matlab-Engr
Command Window
lab_contac...
er_Schools...
er_Schools...
ting an As...

```
Figure \(\PageIndex{i}\): How to Change MATLAB's Working Directory
```



Opening and Saving m-Files in Matlab

New Script
New Live Script
New
Open
Find Files
Compare
Import Data
Save Workspace
New Variable
Open Variable
Clear Workspace
FILE
VARIABLE
CODE
SIMULINK
ENVIRONMENT
RE
Users
CareyToshiba
Documents
College
Oxnard
2018-1Spring
Matlab-Engr
Command Window
Editor - Untitled
EDI...
PU...
VI...
HW2_4.m
Untitled
Editor - C:\Users\CareyToshiba\Documents\College\
EDITOR
PUBLISH
VIEW
HW2_4_CAS.m
Untitled

Open a new script (m-file) with either of these icons.

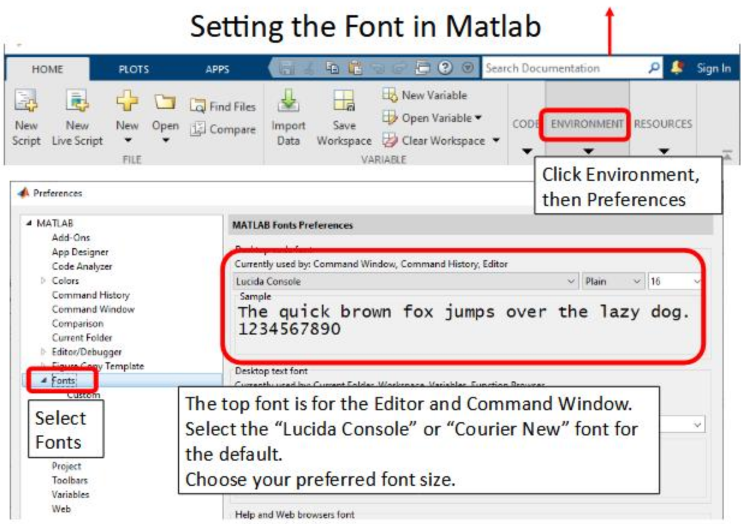
```
>> pwd
ans =
    'C:\Users\CareyToshiba\Documents\College\Oxnard\2018-
>>
```

Enter your code. Then “Save As” with a descriptive file name.

```
%% Problem 2.4
%a)
r=5
area= pi*r^2          % 78.5398
%b)
r=10
surface_area= 4*pi*r^2  % 1.25
```

A MATLAB live script (a new feature) is an interactive document that combines MATLAB code with embedded output, formatted text, equations, and images in a single environment. See the Matlab help documentation for more information.

Figure \(\PageIndex{i}\): Opening and Saving m-Files in MATLAB User Interface's Ini



Setting the Font in Matlab

HOME
PLOTS
APPS
Search Documentation
Sign In
New Script
New Live Script
New
Open
Find Files
Compare
Import Data
Save Workspace
New Variable
Open Variable
Clear Workspace
FILE
VARIABLE
CODE
ENVIRONMENT
RESOURCES
Preferences
MATLAB
Add-Ons
App Designer
Code Analyzer
Colors
Command History
Command Window
Comparison
Current Folder
Editor/Debugger
Figure Copy Template
Fonts
Custom
Project
Toolbars
Variables
Web
MATLAB Fonts Preferences
Desktop text font
Help and Web browsers font

Click Environment, then Preferences

Currently used by: Command Window, Command History, Editor

Lucida Console | Plain | 16

Sample

The quick brown fox jumps over the lazy dog.
1234567890

Select Fonts

The top font is for the Editor and Command Window. Select the “Lucida Console” or “Courier New” font for the default. Choose your preferred font size.

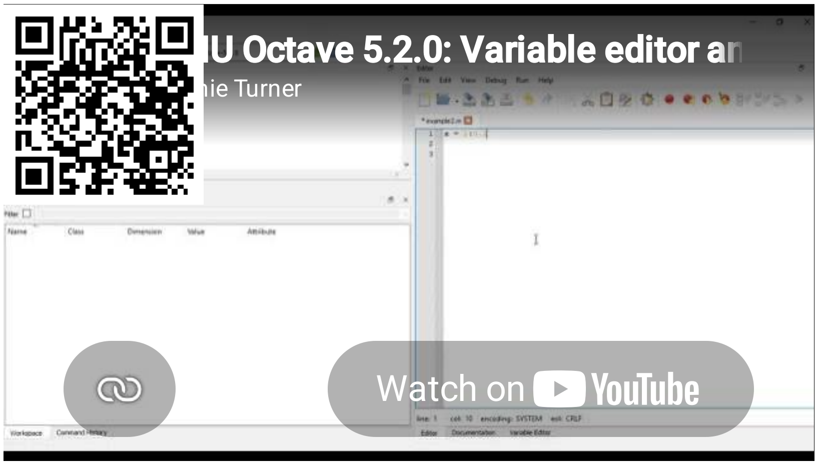
Figure \(\PageIndex{i}\): Setting the Font in MATLAB

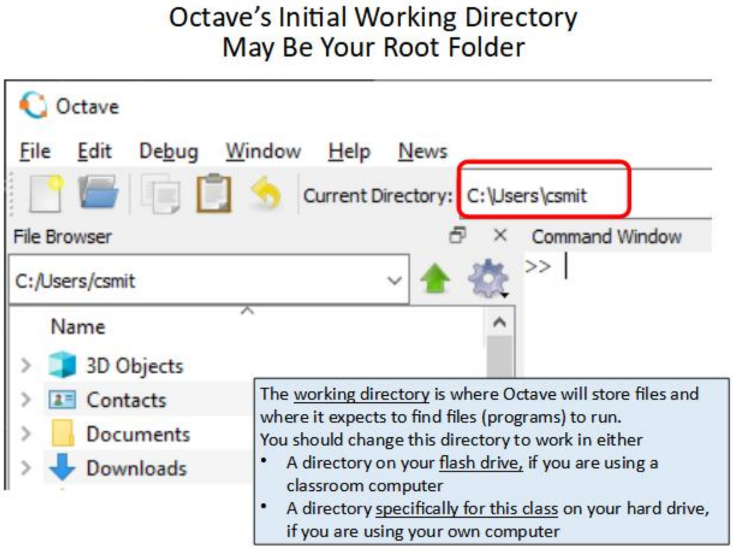
This page titled 1.4: MATLAB User Interface is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

1.5: Octave User Interface

By Carey A. Smith

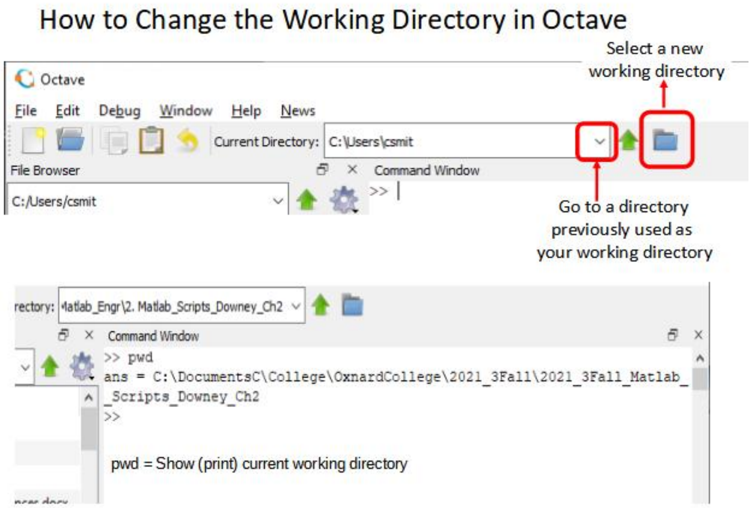
Watch this video. Then read the information below.


U Octave 5.2.0: Variable editor ar
ie Turner
Watch on YouTube


Octave's Initial Working Directory
May Be Your Root Folder
Octave
File
Edit
Debug
Window
Help
News
Current Directory:
C:\Users\csmit
File Browser
Command Window
C:/Users/csmit
>>
Name
3D Objects
Contacts
Documents
Downloads

The working directory is where Octave will store files and where it expects to find files (programs) to run.
You should change this directory to work in either
- A directory on your flash drive, if you are using a classroom computer
- A directory specifically for this class on your hard drive, if you are using your own computer

Figure \(\PageIndex{i}\): Octave User Interface's Initial Working Directory



How to Change the Working Directory in Octave

Select a new working directory

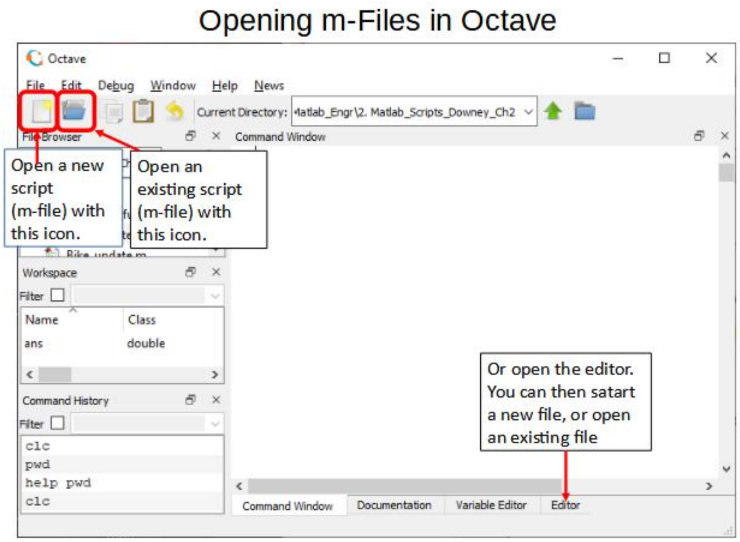
Go to a directory previously used as your working directory

Octave
File Edit Debug Window Help News
Current Directory: C:\Users\csmit
File Browser
C:/Users/csmit
Command Window
>> |
rectory: Matlab_Engr\2. Matlab_Scripts_Downey_Ch2
Command Window

```
>> pwd
ans = C:\DocumentsC\College\OxnardCollege\2021_3Fall\2021_3Fall_Matlab_
_Scripts_Downey_Ch2
>>
```

pwd = Show (print) current working directory

```
Figure \(\PageIndex{i}\): How to Change Octave's Working Directory
```



Opening m-Files in Octave

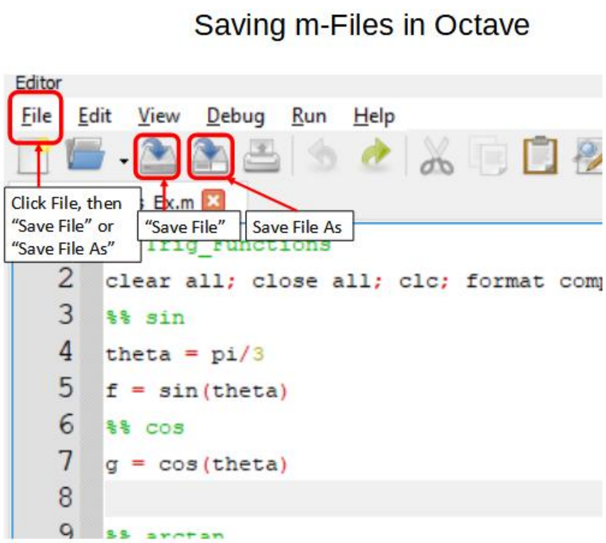
Open a new script (m-file) with this icon.

Open an existing script (m-file) with this icon.

Or open the editor. You can then satart a new file, or open an existing file

Octave
File Edit Debug Window Help News
Current Directory: Matlab_Engr\2. Matlab_Scripts_Downey_Ch2
File Browser
Command Window
Workspace
Filter
Name
Class
ans
double
Command History
Filter
clc
pwd
help pwd
clc
Command Window
Documentation
Variable Editor
Editor

```
Figure \(\PageIndex{i}\): Opening m-Files in Octave
```



Saving m-Files in Octave

Editor
File
Edit
View
Debug
Run
Help
Ex.m

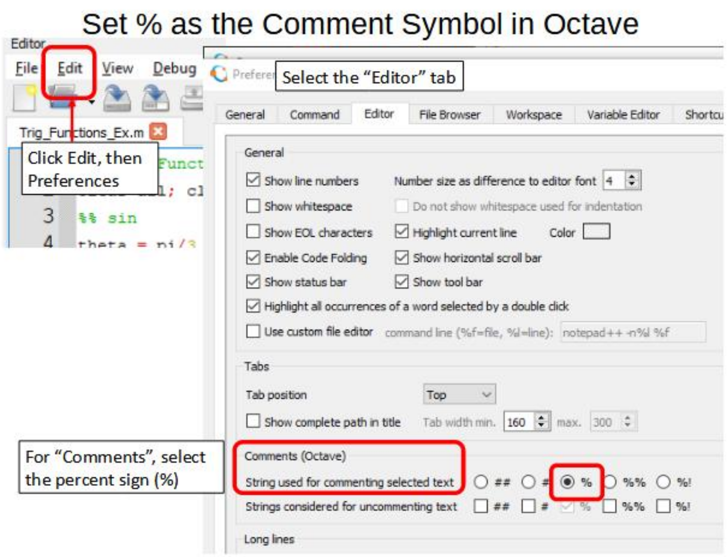
Click File, then "Save File" or "Save File As"

"Save File"

Save File As

```
Trig_Functions
clear all; close all; clc; format com
%% sin
theta = pi/3
f = sin(theta)
%% cos
g = cos(theta)

%% arctan
```

Figure \(\PageIndex{i}\): Saving m-Files in Octave



Set % as the Comment Symbol in Octave

Editor
File
Edit
View
Debug
Trig_Functions_Ex.m
Prefere
General
Command
Editor
File Browser
Workspace
Variable Editor
Shortcu

Click Edit, then Preferences

Select the "Editor" tab

For "Comments", select the percent sign (%)

General

☑ Show line numbers   Number size as difference to editor font 4

☐ Show whitespace   ☐ Do not show whitespace used for indentation

☐ Show EOL characters   ☑ Highlight current line   Color

☑ Enable Code Folding   ☑ Show horizontal scroll bar

☑ Show status bar   ☑ Show tool bar

☑ Highlight all occurrences of a word selected by a double click

☐ Use custom file editor   command line (%f=file, %l=line): notepad++ -n%l %f

Tabs

Tab position Top

☐ Show complete path in title   Tab width min. 160 max. 300

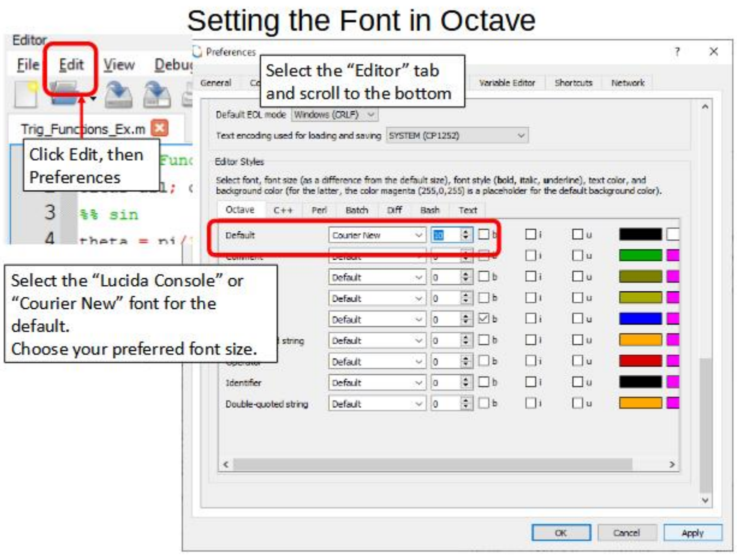
Comments (Octave)

String used for commenting selected text ○ ## ○ # ● % ○ %% ○ %!

Strings considered for uncommenting text ☐ ## ☐ # ☑ % ☐ %% ☐ %!

Long lines

Figure \(\PageIndex{i}\): Set % as the comment symbol in Octave



Setting the Font in Octave

Click Edit, then Preferences

Select the "Editor" tab and scroll to the bottom

Select the "Lucida Console" or "Courier New" font for the default.
Choose your preferred font size.

Default EOL mode: Windows (CRLF)

Text encoding used for loading and saving: SYSTEM (CP1252)

Editor Styles

Select font, font size (as a difference from the default size), font style (bold, italic, underline), text color, and background color (for the latter, the color magenta (255,0,255) is a placeholder for the default background color).

Default: Courier New, 10

Editor
File
Edit
View
Debug
Trig_Functions_Ex.m
3
%% sin
4
theta = pi/
Preferences
General
Variable Editor
Shortcuts
Network
Octave
C++
Perl
Batch
Diff
Bash
Text
Identifier
Double-quoted string
OK
Cancel
Apply

```
Figure \(\PageIndex{i}\): Setting the Font in Octave
```

This page titled 1.5: Octave User Interface is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

1.6: Variables and Assignment Statements

Of course, MATLAB is good for more than just evaluating expressions. One of the features that makes MATLAB more powerful than a calculator is the ability to give a name to a value. A named value is called a *variable*.

MATLAB comes with a few predefined variables. For example, the name `pi` refers to the mathematical quantity π, which is approximately this:

```
>> pi
ans = 3.1416
```

And if you do anything with complex numbers, you might find it convenient that both `i` and `j` are predefined as the square root of -1.

You can use a variable name anywhere you can use a number—for example, as an operand in an expression,

```
>> pi * 3^2
ans = 28.2743
```

or as an argument to a function:

```
>> sin(pi/2)
ans = 1
```

Whenever you evaluate an expression, MATLAB assigns the result to a variable named `ans`. You can use `ans` in a subsequent calculation as shorthand for “the value of the previous expression.”

```
>> 3^2 + 4^2
ans = 25

>> sqrt(ans)
ans = 5
```

But keep in mind that the value of `ans` changes every time you evaluate an expression.

Assignment Statements

You can create your own variables, and give them values, with an *assignment statement*. The assignment operator is the equals sign (`=`), used like so:

```
>> x = 6 * 7
x = 42
```

This example creates a new variable named `x` and assigns it the value of the expression `6 * 7`. MATLAB responds with the variable name and the computed value.

There are a few rules when assigning variables a value. In every assignment statement, the left side has to be a legal variable name. The right side can be any expression, including function calls. Almost any sequence of lower- and uppercase letters is a legal variable name. Some punctuation is also legal, but the underscore (`_`) is the only commonly used non-letter. Numbers are fine, but not at the beginning. Spaces are not allowed. Variable names are *case sensitive*, so `x` and `X` are different variables.

Let’s look at some examples of assignment statements.

```
>> fibonacci0 = 1;

>> LENGTH = 10;

>> first_name = 'bob'
first_name = 'bob'
```

The first two examples demonstrate the use of the semicolon, which suppresses the output from a command. In this case MATLAB creates the variables and assigns them values but displays nothing.

The third example demonstrates that not everything in MATLAB is a number. A sequence of characters in single quotes is a *string*.

Although `i` , `j` , and `pi` are predefined, you are free to reassign them. It's common to use `i` and `j` for other purposes, but it's rare to assign a different value to `pi` .

Variables in the Workspace

When you create a new variable, it appears in the Workspace window and is added to the *workspace*, which is a set of variables and their values.

The `who` command prints the names of the variables in the workspace:

```
>> x = 5;
>> y = 7;
>> z = 9;
>> who

Your variables are:

x  y  z
```

The `clear` command removes specified variables from the workspace:

```
>> clear x
>> who

Your variables are:

y z
```

But be careful: if you don't specify any variables, `clear` removes them all.

To display the value of a variable, you can use the `disp` function:

```
>> disp(z)
     9
```

but it's easier to just type the variable name:

```
>> z
z = 9
```

Now that you've seen how to use them, let's take a step back and think about why we'd use variables.

Why Variables?

There are a number of reasons to use variables. A big one is to avoid recomputing a value you use repeatedly. For example, if your computation uses e frequently, you might want to compute it once and save the result.

```
>> e = exp(1)
e = 2.7183
```

Variables also make the connection between the code and the underlying mathematics more apparent. If you're computing the area of a circle, you might want to use a variable named `r`:

```
>> r = 3
r = 3

>> area = pi * r^2
area = 28.2743
```

That way, your code resembles the familiar formula $a = \pi r^2$.

You might also use a variable to break a long computation into a sequence of steps. Suppose you're evaluating a big, hairy expression like this:

```
p = ((x - theta) * sqrt(2 * pi) * sigma)^-1 * ...
exp(-1/2 * (log(x - theta) - zeta)^2 / sigma^2)
```

You can use an ellipsis to break the expression into multiple lines. Just enter `...` at the end of the first line and continue on to the next. But often it's better to break the computation into a sequence of steps and assign intermediate results to variables:

```
shiftx = x - theta
denom = shiftx * sqrt(2 * pi) * sigma
temp = (log(shiftx) - zeta) / sigma
exponent = -1/2 * temp^2
p = exp(exponent) / denom
```

The names of the intermediate variables explain their role in the computation: `shiftx` is the value of `x` shifted by `theta`, it should be no surprise that `exponent` is the argument of `exp`, and `denom` ends up in the denominator. Choosing informative names makes the code easier to read and understand, which makes it easier to debug.

This page titled 1.6: Variables and Assignment Statements is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **1.3: Variables** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

1.6.1: More About Valid Variable Names

1.6.1.1: Variable Name Rules

MATLAB variable names must start with a letter. The rest of the name can be letters, numbers or the underscore symbol. The letters can be upper or lower case. These are valid variable names:

- b
- B
- Basketball
- Basket_Ball
- basket5ball
- basketBall_12345

These are not valid names:

- Basket-Ball
- Basket_Ball!
- Basket?Ball
- Basket Ball

Descriptive names are encouraged, in order to make the code readable. So 'pressure' and 'press' are better names than just 'p'. Names are allowed to be very long, but they should not be more than a quarter of the command window, so keep them to less than 20 letters, unless there is a very good reason to make them longer. (The maximumn allowed name length is 63 characters.)

Sometimes, you may have 2 version of the same quantity, such as a distance in both inches and centimeters. A good practice is to give them related names with identifiers, such as dist_inch and dist_cm, so that you always know which units you are using.

1.6.1.2: Variable Names to Avoid

Don't use the name of a function. For example, don't use these names:

sqrt, sin, cos, log10, sum.

Also, don't use 2 names that only differ for upper vs. lower case letters; that can lead to confusion. For example, having 2 variables named kg and Kg should be avoided.

? Exercise 1.6.1.1 Good Variable Names

Which of the following are good variable names? Make a decision about each before looking at the answers.

You may enter each in the command window by setting the name = 1, in order to see if it is valid.

You may use the exist() function to see if it is a function name.

sind

sinx

A-very_long_name

Even_123_longer456_name_9876543210

important!

machuPichu

_next

CC BY-NC-SA

1one1

Answer

sind -- a function name--don't use it

sinx -- valid

A-very_long_name -- not valid becuz of the -, which is interpreted as a minus sign

Even_123_longer456_name_9876543210 -- valid

important! -- nor valid becuz of the !

machuPichu

_next -- not valid. It can't start with an underscore

1one1 -- not valid. It can't start with a number

.

1.6.1.3: MATLAB keywords

It is not allowed to use MATLAB keywords. These are words that MATLAB reserves for program structure. This is the list of keywords:

'break'
'case'
'catch'
'classdef'
'continue'
'else'
'elseif'
'end'
'for'
'function'
'global'
'if'
'otherwise'
'parfor'
'persistent'
'return'
'spmd'
'switch'
'try'
'while'

This page titled 1.6.1: More About Valid Variable Names is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

1.7: Clearing the Workspace, Readability Tips

1.7.1: Clearing the workspace

Generally, it's a good idea to use the following code when starting a MATLAB session or script file:

```
format compact;  % Don't insert blank lines on the console--saves screen space
clear all;       % Remove all previously defined variables
close all;       % Close all plots (figures)
clc;             % Clear the console (screen)
clear functions; % If you created a name collision by creating a function with the same name as a built-in function,
% then delete your function file and use "clear functions" to erase that reference from the computer memory.
```

Your can combine them all on 1 line like this:

```
format compact; clear all; close all; clc; clear functions;
```

1.7.2: Readability

This example shows that you don't have to put in spaces between numbers and operators, but spaces help readability. Try these two expressions:

1+2+3+4+5+6

1 + 2 + 3 + 4 + 5 + 6

Both expressions give the same answer.

These other arithmetic operators are clear:
2*3 - 4/5
2 raised to the 8th power is entered with the ^ symbol, called "caret"
2^8
The algebra order of operations is applied in MATLAB instructions.
Use parentheses when you need a different order of operations.
Parentheses can help readability even when not required.

```
2*(3-4)/5
```

This page titled 1.7: Clearing the Workspace, Readability Tips is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

1.8: Errors

Every error is a learning opportunity. Whenever you learn a new feature, you should try to make as many errors as possible, as soon as possible. When you make deliberate errors, you see what the error messages are. Later, when you make accidental errors, you'll know what the messages mean.

Let's look at some common errors. A big one for beginners is leaving out the `*` for multiplication, as in this example:

```
area = pi r^2
```

This code should produce the following error message:

```
area = pi r^2
          |
Error: Invalid expression. Check for missing multiplication
operator, missing or unbalanced delimiters, or other syntax
error. To construct matrices, use brackets instead of parentheses.
```

The message indicates that the expression is invalid and suggests several things that might be wrong. In this case, one of its guesses is right: we're missing a multiplication operator.

Another common error is to leave out the parentheses around the arguments of a function. For example, in math notation it's common to write something like $\sin \pi$, but in MATLAB if you write

```
sin pi
```

you should get the following error message:

```
Undefined function ’sin’ for input arguments of type 'char'.
```

The problem is that when you leave out the parentheses, MATLAB treats the argument as a string of characters (which have type `'char'`). In this case the error message is helpful, but in other cases the results can be baffling. For example, if you call `abs`, which computes absolute values, and forget the parentheses, you get a surprising result:

```
>> abs pi
ans =  112   105
```

I won't explain this result; for now, I'll just suggest that you should put parentheses around arguments.

Here's another common error. If you were translating the mathematical expression

$$\frac{1}{2\sqrt{\pi}} \tag{1.8.1}$$

into MATLAB, you might be tempted to write this:

```
1 / 2 * sqrt(pi)
```

But that would be wrong because of the order of operations. Division and multiplication are evaluated from left to right, so this expression would multiply `1/2` by `sqrt(pi)`.

To keep `sqrt(pi)` in the denominator, you could use parentheses,

```
1 / (2 * sqrt(pi))
```

or make the division explicit,

```
1 / 2 / sqrt(pi)
```

The last two examples bring us to the Second Theorem of Debugging:

> *The only thing worse than getting an error message is not getting an error message.*

Beginning programmers often hate error messages and do everything they can to make the messages go away. Experienced programmers know that error messages are your friend. They can be hard to understand, and even misleading, but it’s worth the effort to understand them.

This page titled 1.8: Errors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **1.4: Errors** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

1.9: Built-in Function Documentation

MATLAB comes with two forms of documentation, `help` and `doc`. The `help` command works in the Command Window; just enter **`help`** followed by the name of a command.

```
>> help sin
```

You should see output like this:

```
sin    Sine of argument in radians.
    sin(X) is the sine of the elements of X.

    See also asin, sind, sinpi.
```

Some documentation uses vocabulary we haven't covered yet. For example, `the elements of X` might not make sense until we get to vectors and matrices a few chapters from now.

The `doc` pages are usually better. If you enter `doc sin`, a browser window appears with more detailed information about the function, including examples of how to use it. The examples often use vectors and matrices, so they may not make sense yet, but you can get a preview of what's coming.

This page titled 1.9: Built-in Function Documentation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **1.5: Documentation** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

1.10: Chapter Review

This chapter provided an overview of the modeling process, including abstraction, analysis and simulation, measurement, and validation.

It also introduced MATLAB, the programming language we'll use to write simulations. So far, we've seen variables and values, arithmetic operations, and mathematical functions.

Here are a few terms from this chapter you might want to remember.

The *interpreter* is the program that reads and executes MATLAB or code. It prints a *prompt* to indicate that it's waiting for you to type a *command*, which is a line of code executed by the interpreter.

An *operator* is a symbol, like `*` or `+`, that represents a mathematical operation. An *operand* is a number or variable that appears in an expression along with operators. An *expression* is a sequence of operands and operators that specifies a mathematical computation and yields a value.

A *function* is a named computation; for example, `log10` is the name of a function that computes logarithms in base 10. A *function call* is a command that causes a function to execute and compute a result. An *argument* is an expression that appears in a function call to specify the value the function operates on.

A *variable* is a named value. An *assignment statement* is a command that creates a new variable (if necessary) and gives it a value. A *workspace* is a set of variables and their values.

Finally, a *string* is a value that consists of a sequence of characters (as opposed to a number).

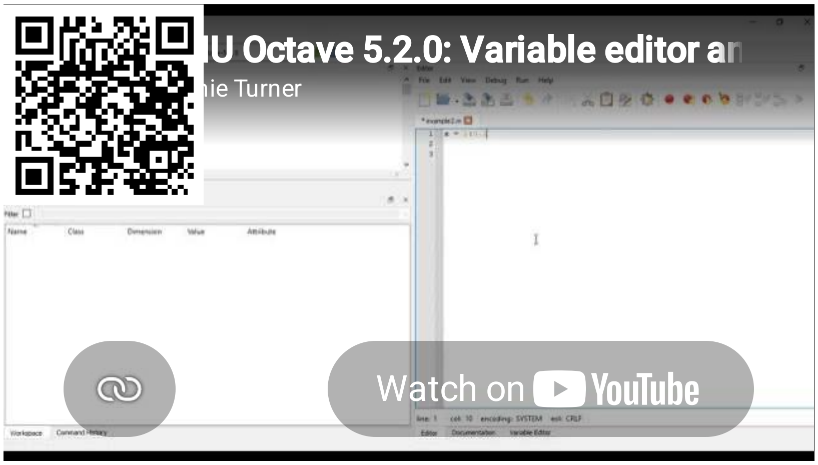
In the next chapter, you'll start writing longer programs and learn about floating-point numbers.

This page titled 1.10: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **1.6: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

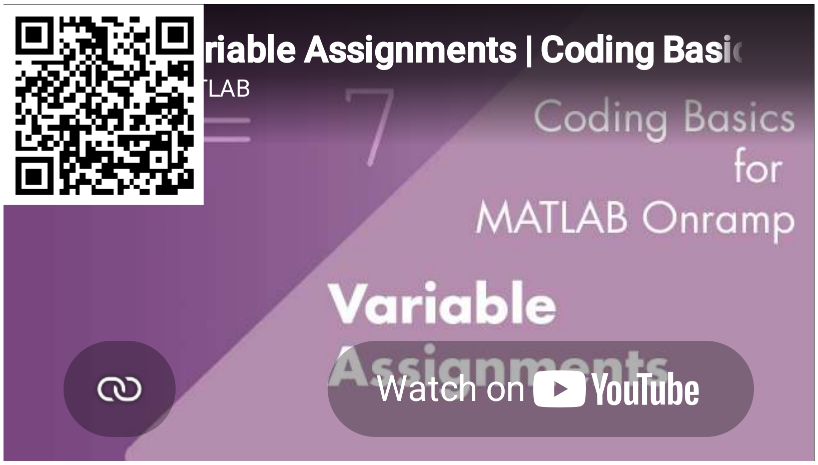
1.11: Chapter 1 videos English and Espanol

English


U Octave 5.2.0: Variable editor ar
ie Turner
Watch on YouTube

.

Variable Assignments | Coding Basics for MATLAB Onramp, Part 2


riable Assignments | Coding Basic
LAB
Coding Basics
for
MATLAB Onramp
Variable
Assignments
Watch on YouTube

Working in the Development Environment

rking in the Development Environ
LAB
The MATLAB
Development Environment
Watch on YouTube

Espanol

Aritmética y variables - 1 - Octave

tmética y variables - 1 - Octave
siored
itmética
Variables
Watch on YouTube

Funciones de Octave - 2 - Octave

nciones de Octave - 2 - Octave
siored
unciones de
Octave
Watch on YouTube

.

https://www.youtube.com/c/MATLABenEspanol

Descubre MATLAB En Español
MATLAB en Español
SUBSCRIBE
youtube.com/MATLABenEspanol
SUSCRÍBETE
MathWorks
0:30 / 0:30

.

This page titled 1.11: Chapter 1 videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

1.12: Exercises

Before you go on, you might want to work on the following exercises.

? Exercise 1.12.1

You might have heard that a penny dropped from the top of the Empire State Building would be going so fast when it hit the pavement that it would be embedded in the concrete or that if it hit a person it would break their skull.

We can test this myth by making and analyzing a model. To get started, we'll assume that the effect of air resistance is small. This will turn out to be a bad assumption, but bear with me.

If air resistance is negligible, the primary force acting on the penny is gravity, which causes the penny to accelerate downward.

If the initial velocity is 0, the velocity after t seconds is at, and the distance the penny has dropped is

$$h = at^2/2$$

Using algebra, we can solve for t:

$$t = \sqrt{2h/a}$$

Plugging in the acceleration of gravity, $a = 9.8\ \mathrm{m/s^2}$, and the height of the Empire State Building, $h = 381\ \mathrm{m}$, we get $t = 8.8\ \mathrm{s}$. Then, computing $v = at$ we get a velocity on impact of $86\ \mathrm{m/s}$, which is about 190 miles per hour. That sounds like it could hurt.

Use MATLAB to perform these computations, and check that you get the same result.

? Exercise 1.12.2

The result in the previous exercise is not accurate because it ignores air resistance. In reality, once the penny gets to about 18 $\mathrm{m/s}$, the upward force of air resistance equals the downward force of gravity, so the penny stops accelerating. After that, it doesn't matter how far the penny falls; it hits the sidewalk at about $18\ \mathrm{m/s}$, much less than $86\ \mathrm{m/s}$.

As an exercise, compute the time it takes for the penny to reach the sidewalk if we assume that it accelerates with constant acceleration $a = 9.8\ \mathrm{m/s^2}$ until it reaches terminal velocity, then falls with constant velocity until it hits the sidewalk.

The result you get is not exact, but it's a pretty good approximation.

This page titled 1.12: Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **1.7: Exercises** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

CHAPTER OVERVIEW

2: MATLAB/Octave Scripts

2.1: How to Create a Script File and Valid File Names
2.2: Your First Script
  2.2.1: .mlx Live Editor Files
2.3: Examples of assignment scripts
2.4: Why Scripts?
2.5: The Fibonacci Sequence
2.6: Floating-Point Numbers
2.7: Scientific Notation for Large and Small Numbers
2.8: Number formats
2.9: Complex Numbers
2.10: Script Comments
2.11: Script Documentation
2.12: Assignment and Equality
2.13: Understanding Editor Warning and Error messages
2.14: Debugging Directory Errors
2.15: Chapter Review
2.16: Chapter 2 Scripts videos English and Espanol
2.17: Exercises
  2.17.1: More Exercises

Scripts are code files. They allow you to save code and to change input values and rerun code without rewriting the code.

This page titled 2: MATLAB/Octave Scripts is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

2.1: How to Create a Script File and Valid File Names

MATLAB file names follow the same rules as variable names:

- MATLAB file names must start with a letter.
- The rest of the name can be letters, numbers or the underscore symbol.
- The letters can be upper or lower case.
- No spaces are allowed.
- No special characters are allowed, except for an underscore.

These are valid file names:

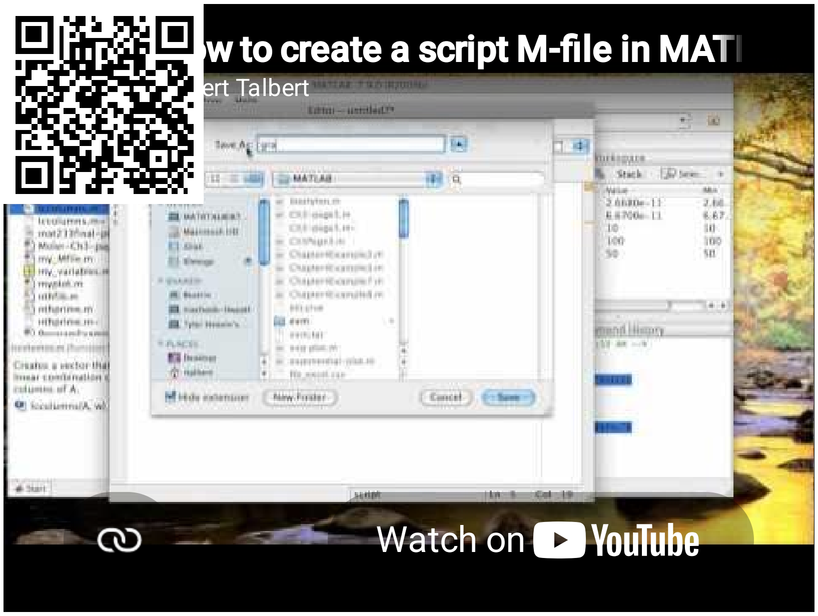
- BrickVolume.m
- My_1st_file.m
- clever_File_123.m

These are not valid file names:

- Brick Volume.m
- 1st_file.m
- gnome@home.m
- Great!.m

These videos show how to create scripts (m-files) in MATLAB and in Octave.

How to create a script M-file in MATLAB, Robert Talbert:



How to create a script M-file in MATLAB, Robert Talbert: [www.youtube.com]

How to Create a MATLAB Script, Mathworks: (The 1st 2 minutes are the most important.)

Mathworks script instructions [www.mathworks.com]

(This also mentions how to create "Live Editor" .mlx files. For most of the assignments in this course, that is more advanced than we need.)

Octave Programming Lesson 4.1 - Script File Basics, Hanshaw Virtual University

tave Programming Lesson 4.1 - Sc
shaw Virtual University
Numerical Analysis Applcations
esson 4.1 - Script File Basics
m. Files 00:45
Octave's Search Path 08:00
Watch on YouTube

How to save a file in Octave:

Octave Programming Lesson 4.1 - Script File Basics, Hanshaw Virtual University [www.youtube.com]

tave Tutorial #8: Writing Scripts a
STEM EDU TV
ting Scripts
& Functions
for Absolute Beginners
Watch on YouTube

.

roducción a scripts en MATLAB
omática EIE
ducción a MATLAB
MATLAB
02
Scripts
Watch on YouTube

Primer script - 3 - Octave

mer script - 3 - Octave
siored
tave
3 Primer script
Watch on YouTube

This page titled 2.1: How to Create a Script File and Valid File Names is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

2.2: Your First Script

A *script* is a file that contains MATLAB code. When you run a script, MATLAB executes the commands in it, one after another, exactly as if you had typed them at the prompt. Scripts are also sometimes called *M-files* because they use the extension *.m*, short for MATLAB.

You can create scripts with any text editor or word processor, but the simplest way is to click the **New Script** button in the upper-left corner of the MATLAB interface, which opens a text editor designed for MATLAB.

To try it out, create a new script and enter the following code:

```
x = 5
```

Then press the **Save** button. A dialog window should appear where you can choose the filename and the folder where your script will go. Change the name to *myscript.m* and save it into any folder you like.

Now click the green **Run** button. You might get a message that says the script is not found in the current folder. If so, click the button that says Change Folder and it should run.

You can also run a script by typing its name in the Command Window and pressing . For example, if you enter `myscript`, MATLAB should execute your script and display the result:

```
>> myscript
x = 5
```

There are a few things to keep in mind when using scripts. First, you should not include the extension *.m* when you run a script. If you do, you'll get an error message like this:

```
>> myscript.m
Undefined variable "myscript" or class "myscript.m".
```

Second, when you name a new script, try to choose something meaningful and memorable. Don't choose a name that's already in use; if you do, you'll replace one of MATLAB's functions with your own (at least temporarily). You might not notice right away, but you might get some confusing behavior later.

Also, the name of the script cannot contain spaces. If you create a file named *my script.m*, MATLAB will complain when you try to run it:

```
>> my script
Undefined function or variable 'my'.
```

It can be hard to remember which folder a script is in. To keep things simple, for now, I suggest putting all of your scripts in one folder.

This page titled 2.2: Your First Script is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

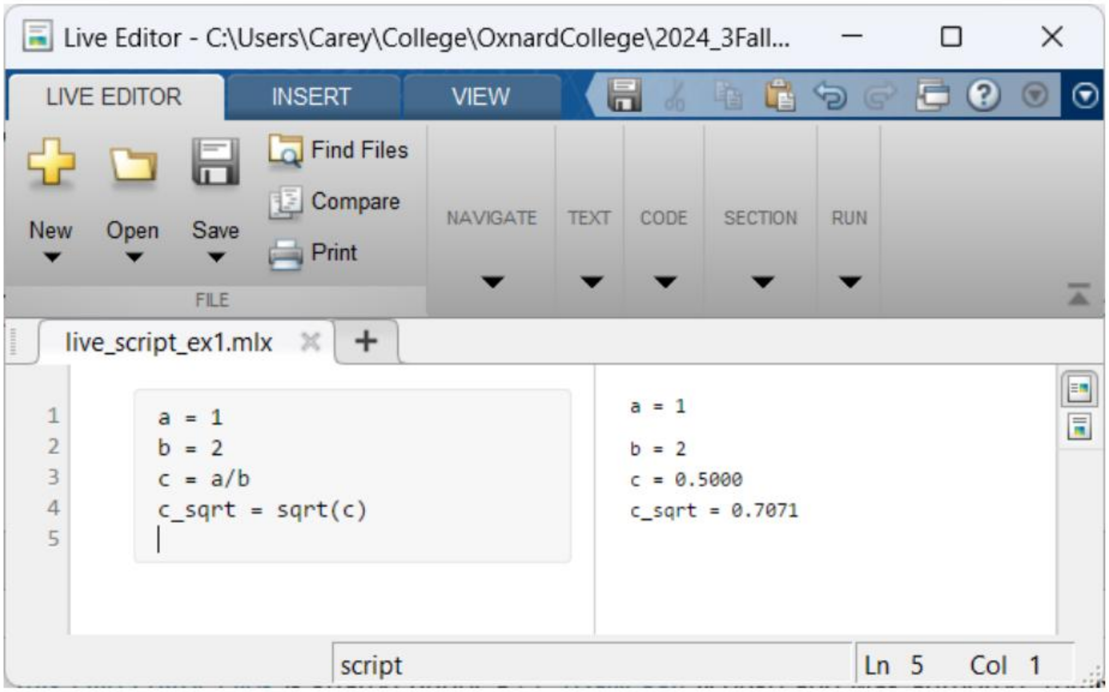
- **2.1: Your First Script** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

2.2.1: .mlx Live Editor Files

This section is optional.

MATLAB has a variation on m-files called a "Live Editor" file. The extension for this file is .mlx. This is not available in Octave.

When a .mlx live script is run, a 2nd column is opened showing the results of the calculations. These results get saved in the .mlx file.


Live Editor - C:\Users\Carey\College\OxnardCollege\2024_3Fall...
LIVE EDITOR
INSERT
VIEW
New
Open
Save
Find Files
Compare
Print
FILE
NAVIGATE
TEXT
CODE
SECTION
RUN
live_script_ex1.mlx
script
Ln 5 Col 1

```
a = 1
b = 2
c = a/b
c_sqrt = sqrt(c)
```

```
a = 1
b = 2
c = 0.5000
c_sqrt = 0.7071
```

In a regular .m file, you can include the results as comments. This can be done in both MATLAB and Octave.

An example of results in listed in comments:

theta = pi/8;
sin(theta) % ans = 0.3827

This is a link to MATLAB help n live scripts: https://www.mathworks.com/help/matlab/matlab_prog/create-live-scripts.html

---

This page titled 2.2.1: .mlx Live Editor Files is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

2.3: Examples of assignment scripts

An m-file for an assignment should have just the code that sets the variables and does the calculations. The teacher will run your code, so the assignment m-file does not need the results of the calculations. If you choose to put in the results, put them in as comments. To create a comment, use the % symbol.

Example 2.3.1 Compute the area of a rectangle

Assignment: Create an m-file that sets the width and height of a rectangle, then computes its area.

Solution

One solution is an m-file with this code:

```
% Compute the area of a rectangle
rect_width  = 3 % cm
rect_height = 5 % cm
rect_area   = rect_width * rect_height
% rect_area = 15
```

.

This next example has 2 files.

The 1st file is a script that does the calculation of the area of an isosceles triangle. It requires that the base and height of the triangle be defined before running the script. These are the 2 preconditions of the script.

The 2nd file is a script that tests the 1st file. This 2nd file defines values to the base and height variables, then it has a line of code with the name of the 1st file. This line runs the 1st file. This 2nd file then defines a second set of values to the base and height variables and has a line of code with the name of the 1st file. This line runs the 1st file with the new values.

Example 2.3.2 Using a test file to run a script with 2 sets of inputs

Assignment: Create an 2 m-files. The 1st m-file computes computes the area of an isosceles triangle. Call this file isosceles_triangle_area.m

It has preconditions that tri_base and tri_height are defined before running the script.

The second script file is a test file. Its defines this set of values:

```
tri_base = 4
tri_height = 3
```

Then it runs the isosceles_triangle_area script by putting the name of the 1st script, "isosceles_triangle_area", on a line of code.

After the area of the first triangle is computed, the test file defines this second set of values:

```
tri_base = 10
tri_height = 50
```

Again, it runs the isosceles_triangle_area script by putting "isosceles_triangle_area" on a line of code.

Solution

The isosceles_triangle_area.m file:

```
% Compute the area of an isocesles triangle
% Preconditions (Inputs):
```

```
% tri_base
% tri_height
% Postcondition (Output)
% tri_area
tri_area = (1/2)*tri_base*tri_height
```

The isosceles_triangle_area_test.m file:

```
% test the isosceles_triangle_area script
% Triangle 1:
% Define the inputs:
tri_base = 4
tri_height = 3
% Compute the area:
isosceles_triangle_area

% Triangle 2:
% Define the inputs:
tri_base = 10
tri_height = 50
% Compute the area:
isosceles_triangle_area
```

.

This page titled 2.3: Examples of assignment scripts is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

2.4: Why Scripts?

There are a few good reasons to use a script. When you're writing more than a couple of lines of code, it might take a few tries to get everything right. Putting your code in a script makes it easier to edit than typing it at the prompt. Likewise, if you're running a script repeatedly, it's much faster to type the name of the script than to retype the code! And you might be able to reuse a script from one project to the next, saving you considerable time across projects.

But the great power of scripts comes with great responsibility: you have to make sure that the code you are running is the code you think you are running. Whenever you start a new script, start with something simple, like `x = 5`, that produces a visible effect. Then run your script and confirm that you get what you expect. When you type the name of a script, MATLAB searches for the script in a *search path*, which is a sequence of folders. If it doesn't find the script in the first folder, it searches the second, and so on. If you have scripts with the same name in different folders, you could be looking at one version and running another.

If the code you are running is not the code you are looking at, you'll find debugging a frustrating exercise! So it's no surprise that this is the Third Theorem of Debugging:

> *Be sure that the code you are running is the code you think you are running.*

Now that you've seen how to write a script, let's use one to do something a little more complicated.

This page titled 2.4: Why Scripts? is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.2: Why Scripts?** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

2.5: The Fibonacci Sequence

The *Fibonacci sequence*, denoted F, is a sequence of numbers where each number is the sum of the previous two. It's defined by the equations $F_1 = 1$, $F_2 = 1$, and, for $i > 2$, $F_i = F_{i-1} + F_{i-2}$. The following expression computes the nth Fibonacci number:

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

We can translate this expression into MATLAB, like this:

```
s5 = sqrt(5);
t1 = (1 + s5) / 2;
t2 = (1 - s5) / 2;
diff = t1^n - t2^n;
ans = diff / s5
```

I use temporary variables like `t1` and `t2` to make the code readable and the order of operations explicit. The first four lines have a semicolon at the end, so they don't display anything. The last line assigns the result to `ans` .

If we save this script in a file named *fibonacci1.m*, we can run it like this:

```
>> n = 10
>> fibonacci1
ans = 55.0000
```

Before calling this script, you have to assign a value to `n` . If `n` is not defined, you get an error:

```
>> clear n
>> fibonacci1
Undefined function or variable 'n'.

Error in fibonacci1 (line 9)
diff = t1^n - t2^n;
```

This script only works if there is a variable named `n` in the workspace; otherwise, you should get an error. MATLAB will tell you what line of the script the error is in and display the line.

Error messages can be helpful, but beware! In this example, the message says the error is in `fibonacci` , but the actual problem is that we have not assigned a value to `n` . And that brings us to the Fourth Theorem of Debugging:

> *Error messages tell you where the problem was discovered, not where it was caused.*

Often you have to work backwards to find the line of code (or missing line) that caused the problem.

This page titled 2.5: The Fibonacci Sequence is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.3: The Fibonacci Sequence** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

2.6: Floating-Point Numbers

In the previous section, the result we computed was `55.0000`. Since the Fibonacci numbers are integers, you might have been surprised to see the zeros after the decimal point.

They are there because MATLAB performs calculations using floating-point numbers. With *floating-point numbers*, integers can be represented exactly, but most fractions cannot.

For example, if you compute the fraction 2/3, the result is only approximate—the correct answer has an infinite number of 6s:

```
>> 2/3
ans = 0.6666
```

It's not as bad as this example makes it seem: MATLAB uses more digits than it shows by default. You can use the `format` command to change the output format:

```
>> format long
>> 2/3
ans = 0.666666666666667
```

In this example, the first 14 digits are correct; the last one has been rounded off.

Large and small numbers are displayed in *scientific notation*. For example, if we use the built-in function `factorial` to compute $100!$, we get the following result:

```
>> factorial(100)
ans = 9.332621544394410e+157
```

The `e` in this notation is *not* the transcendental number known as e; it's just an abbreviation for "exponent." So this means that $100!$ is approximately 9.33×10^{157}. The exact solution is a 158-digit integer, but with double-precision floating-point numbers, we only know the first 16 digits.

You can enter numbers using the same notation.

```
>> speed_of_light = 3.0e8
speed_of_light = 300000000
```

Although the floating-point format can represent very large and small numbers, there are limits. The predefined variables `realmax` and `realmin` contain the largest and smallest numbers MATLAB can handle.

```
>> realmax
ans = 1.797693134862316e+308

>> realmin
ans = 2.225073858507201e-308
```

If the result of a computation is too big, MATLAB "rounds up" to infinity.

```
>> factorial(170)
ans = 7.257415615307994e+306

>> factorial(171)
ans = Inf
```

Division by zero also returns `Inf`.

```
>> 1/0
ans = Inf
```

For operations that are undefined, MATLAB returns `NaN`, which stands for “not a number.”

```
>> 0/0
ans = NaN
```

This page titled 2.6: Floating-Point Numbers is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.4: Floating-Point Numbers** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

2.7: Scientific Notation for Large and Small Numbers

It is common in the USA to enter numbers >= 1000 using commas. However, commas are not used that way in MATLAB and Octave. As you will read in the chapter on vectors, commas are used to separate elements of a vector. One way to enter large numbers is to simply leave out the commas, as shown in these examples:

```
a = 1000
```

```
b = 54321
```

If you enter b with a comma:

```
b = 54,321
```

the result is is 2 separate numbers:

```
b =    54
ans =   321
```

That is not what is intended.

According to the World Bank, the 2023 population of Brazil is 215,313,498

(World Bank, Brazil Population)

In MATLAB or Octave, this can be entered as:

```
Brazil_population= 215313498
```

Alternatively this can be entered using scientific notation (also called exponent notation):

```
Brazil_population= 215.313498E6 % The "E" indicates
```

The speed of light in vacuum can be written (approximately) as

```
c = 3.0E8 % m/s
```

The "E" can be either upper or lower case:

```
c = 3.0e8 % m/s
```

Similar notation can be used for very small numbers. For example the wavelength of a red LED might be 650 nm

```
red_wavelength = 650e-6 % meters
```

This page titled 2.7: Scientific Notation for Large and Small Numbers is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

2.8: Number formats

A straight-forward way to learn MATLAB's number formats is to try them and see their effects.

? Exercise 2.8.1 Number formats

Create a MATLAB .m or .mlx file with the following.

Assign this vector:
a = [-1/3, 0, 1/3, 2/3]

Enter each of the formats below, followed by the variable 'a', in order to see the effect of the format on a number.

format long

a

format bank

a

format short e

a

format long e

a

format short eng

a
format long eng

a

format short g

a

format long g

a

format rat % Ratio

a

format short % Reset to the default format.

Answer

> The effect of 'format long'
> a
>
> Result: -0.333333333333333 0 0.333333333333333 0.666666666666667

A more complete explanation is at: Mathworks format help [www.mathworks.com]

.

Exercise 2.8.1 Fibonnaci_2

A. Create an m-file script for the Fibonnaci example in this section.

Call your file Fibonnaci_2_KennedyJ.m, but replace KennedyJ with your last name and 1st initial.

B. Start with Fibonnaci code of section 2.6.

Add this documentation to the top of the script file:

% Set n = the index of particular Fibonocci number to be computed (precondition)

C. The division in last line of the code can result in a number that not an exact integer.

This can be demonstrated changing the the format to display about 15 decimal places with this line of code:

format long

Set n = 10 and run the script. The result will be

ans = 55.000000000000014

Fix this by changing the last line to be:

ans = round(diff / s5)

D. Test your script with n = 4, n = 5, and n =10

Note: n is a precondition for this script, so n needs to be in the command window, before running the the script.

This allows n to be set to various different value without changing the script.

Answer

Add texts here. Do not delete this text first.

.

This page titled 2.8: Number formats is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

2.9: Complex Numbers

By Carey A. Smith

The imaginary unit number is i.

i = sqrt(-1) or i^2 = -1
In engineering, j is often used instead of i for the imaginary unit number.
In MATLAB, you can use either i or j.
Therefore, i and j should not be used as variable names. Instead, use ii or jj or another letter.
In MATLAB, 2*i can be written as 2i, 3*i as 3i, etc. Try this.
A complex number has both a real part and an imaginary part.

✓ Example 2.9.1 Complex Numbers 1

a = 2 + i
b = 1 - 4i
c = -2 + 3j

Solution

Add example text here.

? Exercise 2.9.1 Complex Numbers real and imag parts, length

% (1 pt) Write a MATLAB m-file script that does the following:
% (1 pt) Define this complex number:
z1 = (2 + 3i)
% (2 pts) Separate the real and imaginary parts like this:
z1_real = real(z1)
z1_imag = imag(z1)

% (1 pt) Compute the conjugate of z1 with this line of code:
z2 = conj(z1) % This changes the sign on the imaginary part
% (1 pt) Compute:
z1*conj(z1) % Note that this = square of the real part + square of the imaginary part

% (1 pt) Compute the length of this vector:
z1_length = sqrt(z1*conj(z1))

Answer

z2 = conj(z1) = 2 - 3i

z1*conj(z1) = 13

z1_length = 3.6056

? Exercise 2.9.2 Plot a Complex Number, version 1

% (3 pts) A complex number can be plotted by letting zx = real(z1) and zy = imag(z1)
% Plot this complex number with these lines of code:
figure;
plot(z1_real,z1_imag,'*')
grid on % Draw x & y grid lines
xlim([-1,4]) % Set the minimum and maximum x values
ylim([-1,4]) % Set the minimum and maximum y values

Answer

Add texts here. Do not delete this text first.

✓ Example $2.9.2$ Plot a Line to a Complex Number

Draw a line from (0,0) to the point with the following code:
hold on % Keep whats been plotted and add to it
plot([0,z1_real],[0,z1_imag])
% [0,z1_real] = x coordinates of start and and end of the line
% [0,z1_imag] = y coordinates of start and and end of the line

Solution

Add example text here.

This page titled 2.9: Complex Numbers is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

2.10: Script Comments

Short, simple programs are easy to read, but as they get bigger and more complex, it gets harder to figure out what they do and how. That’s what comments are for.

A *verbatim* is a line of text added to a program to explain how it works. It has no effect on the execution of the program; it is there for human readers. Good comments make programs more readable; bad comments are useless at best and misleading at worst.

To write a comment, you use the percent symbol (%) followed by the text of the comment.

```
>> speed_of_light = 3.0e8     % meters per second
speed_of_light = 300000000
```

The comment runs from the percent symbol to the end of the line. In this case it specifies the units of the value. In an ideal world, MATLAB would keep track of units and propagate them through the computation, but for now that burden falls on the programmer.

Avoid comments that are redundant with the code:

```
>> x = 5        % assign the value 5 to x
```

Good comments provide additional information that’s not in the code, like units in the example above, or the meaning of a variable:

```
>> p = 0         % position from the origin in meters
>> v = 100       % velocity in meters / second
>> a = -9.8      % acceleration of gravity in meters / second^2
```

If you use longer variable names, you might not need explanatory comments, but there’s a trade-off: longer variable names are clearer, but longer code can become harder to read. Also, if you’re translating from math that uses short variable names, it can be useful to make your program consistent with your math.

This page titled 2.10: Script Comments is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.5: Comments** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

2.11: Script Documentation

Every script should provide *documentation*, which is a comment that explains what the script does and what its requirements are.

For example, I might put something like this at the beginning of *fibonacci1.m*:

```
% Computes a numerical approximation of the nth Fibonacci number.
% Precondition: you must assign a value to n before running this script.
% Postcondition: the result is stored in ans.
```

A *precondition* is something that must be true when the script starts in order for it to work correctly. A *postcondition* is something that will be true when the script completes.

If there is a comment at the beginning of a script, MATLAB assumes it's the documentation for the script. So if you type `help fibonacci1`, you get the contents of the comment (without the percent signs).

```
>> help fibonacci1
  Computes a numerical approximation of the nth Fibonacci number.
  Precondition: you must assign a value to n before running this script.
  Postcondition: the result is stored in ans.
```

That way, scripts that you write behave just like predefined scripts. You can even use the `doc` command to see your comment in the Help Window.

This page titled 2.11: Script Documentation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.6: Documentation** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

2.12: Assignment and Equality

For beginning programmers, a common source of confusion is assignment and the use of the equals sign.

In mathematics, the equals sign means that the two sides of the equation have the same value. In MATLAB, an assignment statement *looks* like a mathematical equality, but it's not.

One difference is that the sides of an assignment statement are not interchangeable. The right side can be any legal expression, but the left side has to be a variable, which is called the *target* of the assignment. So this is legal:

```
>> y = 1;
>> x = y + 1
x = 2
```

But this is not:

```
>> y + 1 = x
 y + 1 = x
       |
Error: Incorrect use of '=' operator.
To assign a value to a variable, use '='.
To compare values for equality, use '=='.
```

In this case the error message is not very helpful. The problem here is that the expression on the left side is not a valid target for an assignment.

Another difference between assignment and equality is that a mathematical equality is true (or false) for all eternity; an assignment statement is temporary. When you assign `x = y + 1`, you get the *current* value of `y`. If `y` changes later, `x` does not get updated.

A third difference is that a mathematical equality is a statement that may or may not be true. In mathematics, $y = y + 1$ is a statement that happens to be false for all values of y. In MATLAB, `y = y + 1` is a sensible and useful assignment statement. It reads the current value of `y`, adds 1, and replaces the old value with the new value.

```
>> y = 1;
>> y = y + 1
y = 2
```

When you read MATLAB code, you might find it helpful to pronounce the equals sign as "gets" rather than "equals." So `x = y + 1` is pronounced "`x` gets the value of `y` plus one."

This page titled 2.12: Assignment and Equality is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.7: Assignment and Equality** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

2.13: Understanding Editor Warning and Error messages

By Carey Smith

In the MATLAB editor, you probably have seen little orange or red lines on the right hand side.
These are there to help you correct your code.
The red lines indicate errors. The code will not run until you fix these. When you have any errors, the little box in the upper right-hand corner will be red.
The orange lines indicate warnings. When you have warnings, but no errors, the little box in the upper right-hand corner will be orange. There are many different kinds of warnings. Read them and decide if they indicate something that needs to be fixed. Some only indicate that a line doesn't have a semi-colon, so that the result will show on the screen, which may be OK, or may be a problem if you have too much showing on the screen.
When you have no warnings and no errors, the little box in the upper right-hand corner will be green.

Read this link to better understand the editor:

Mathworks error help [www.mathworks.com]

This page titled 2.13: Understanding Editor Warning and Error messages is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

2.14: Debugging Directory Errors

Users sometimes correct code errors, but the code still does not run. While this often happens when there are more than 1 errors or when the correction is either not correct or not complete. But sometimes the correction is complete and the new code is fine, but the code still does not run.
Here are some possible causes and solutions.

1. When you have 2 files (a test script and the original script or function), in order for the test script to call (run) the original script or function, both of those files need to be in the working directory (folder). (The words "directory" and "folder" are synonyms in this text.)
1.a. You need to know which directory is being used by the MATLAB or Octave workspace. When you first install Octave or MATLAB, the default directory may be in the Octave or MATLAB directory. You should create a separate directory under "documents" for each course or project.
Then you need to change your your workspace to use that directory. See sections 2.1 or 2.2 on how to change the directory you are operating in.

You can check which directory is being used either by looking at the path above the command window or entering this command: pwd (=print current working directory).

1b. By default, the editor is likely storing your files in the downloads directory or in the C:\Documents directory. In the editor, you use "save as" to save the code files in the directory for your course or project.

2. A second problem is that after you correct errors, it is possible that Octave or MATLAB still remembers and uses the older version. This seems to happen more often with Octave, but it can happen with MATLAB, too. You can test if it is running your old code or your new code by adding an extra line of code such as `xyz = 123` . If that does display, then it is running the new code. If that doesn't display, then an older version of the code is being run.

There are 2 possible solutions to this.
2a. The easiest thing is to make sure the file is saved before you run it. But this may not solve the problem.
2b. The surest thing to do is to use "clear all". This not only clears any variables, but it also clears any references to files you were using. Then when you rerun your code, it will use the current version.

Of course, you may still have actual errors that need correcting.

This page titled 2.14: Debugging Directory Errors is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

2.15: Chapter Review

This chapter presented scripts and suggested reasons to use them. We computed elements of a Fibonacci sequence, but because we used floating-point numbers, the results were sometimes only approximate. And we saw how to add comments to a program to document what it does and explain how it works.

Here are some terms from this chapter you might want to remember.

An *M-file* is a file that contains a *script*, which is a sequence of MATLAB/Octave commands. The *search path* is the list of folders where the interpreter looks for M-files.

A *precondition* is something that must be true when the script starts in order for it to work correctly; a *postcondition* is something that will be true when the script completes.

The *target* of an assignment statement is the variable on the left side.

Floating-point is a way to represent and store numbers in a computer. is a format for typing and displaying large and small numbers; for example, `3.0e8` represents 3.0×10^8 or 300,000,000.

A *verbatim* is part of a program that provides additional information about the program, but does not affect its execution.

In the next chapter, you'll learn how to write programs that perform repetitive tasks using loops.

This page titled 2.15: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.8: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

2.16: Chapter 2 Scripts videos English and Espanol

English

How to Create a MATLAB Script (Mathworks) (Link to a page that has an embedded video)

How to create a MATLAB script 1 [www.mathworks.com]

How to create a script M-file in MATLAB (R. Talbert)

w to create a script M-file in MAT
ert Talbert
Watch on YouTube

tave Programming Lesson 4.1 - S
shaw Virtual University
Numerical Analysis Applcations
esson 4.1 - Script File Basics
m. Files 00:45
Octave's Search Path 08:00
Watch on YouTube

How to Create a MATLAB Script, Mathworks: (The 1st 2 minutes are the most important. This also mentions how to create "Live Editor" .mlx files. For most of the assignments in this course, that is more advanced than we need.)

Introduction to Octave [www.mathworks.com]

Intro to complex numbers (Khan Academy) (Link to a page that has an embedded video)

Complex Numbers, Khan academy [www.khanacademy.org]

Espanol

roducción a scripts en MATLAB
omática EIE
ducción a MATLAB
MATLAB
02
Scripts
Watch on YouTube

Primer script - 3 - Octave

mer script - 3 - Octave
siored
3 Primer script
Watch on YouTube

..

This page titled 2.16: Chapter 2 Scripts videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

2.17: Exercises

Before you go on, you might want to work on the following exercises.

Exercise 2.1

To test your understanding of assignment statements, write a few lines of code that swap the values of `x` and `y`. Put your code in a script called *swap.m* and test it.

If it works correctly, you should be able to run it like this:

```
>> x = 1, y = 2
x = 1
y = 2

>> swap

>> x, y
x = 2
y = 1
```

Exercise 2.2

Imagine that you are the operator of a bike-share system with two locations: Boston and Cambridge.

You observe that every day 5 percent of the bikes in Boston are dropped off in Cambridge, and 3 percent of the bikes in Cambridge get dropped off in Boston. At the beginning of the month, there are 100 bikes at each location.

Write a script called *bike_update.m* that updates the number of bikes in each location from one day to the next. The precondition is that the variables `b` and `c` contain the number of bikes in each location at the beginning of the day. The postcondition is that `b` and `c` have been modified to reflect the net movement of bikes.

To test your program, initialize `b` and `c` at the prompt and then execute the script. The script should display the updated values of `b` and `c`, but not any intermediate variables.

Remember that bikes are countable things, so `b` and `c` should always be integer values. You might want to use the `round` function to compute the number of bikes that move each day.

If you execute your script repeatedly, you can simulate the passage of time from day to day (you can repeat a command by pressing the arrow and then).

What happens to the bikes? Do they all end up in one place? Does the system reach an equilibrium, does it oscillate, or does it do something else?

In the next chapter, we will see how to execute your script automatically and how to plot the values of `b` and `c` over time.

This page titled 2.17: Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.9: Exercises** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

2.17.1: More Exercises

? Exercise 2.17.1.1 Volume of a Cobblestone

A cobblestone (guijarro) is shoebox-shaped.

Create a .m-file (a Matlab script) to compute the volume of a cobblestone.

Set these variables:

cobblestone_length = 9

cobblestone_width = 5

cobblestone_height = cobblestone_width

Then use these variables to compute the volume of this cobblestone.

Test your .m file.

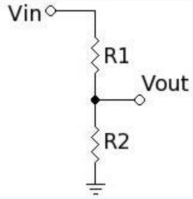
Answer

The answer is not given here.

.

? Exercise 2.17.1.2 Voltage divider

This is a schematic of a voltage divider.



The formula is: Vout = Vin*(R2/(R1+R2))

(1 pt) Write a MATLAB m-file script that computes the output of a voltage divider.

Your file name should be: Voltage_divider_Name.m, where you replace "Name" with your name.

(1 pt) The following variables are set outside of your script before running your script. They are preconditions (inputs) to your script file. Your file script should have these inputs documented by the following comments:

```
% Vin = the input voltage
% R1 = resistance of the 1st resistor
% R2 = resistance of the 2nd resistor
```

(1 pt) In your script, also specify this post condition (output) as a comment:

```
% Vout = the output voltage.
```

(2 pts) In your script calculate the output voltage with this formula:

```
Vout = Vin*(R2/(R1+R2))
```

(6 pts) Write a 2nd MATLAB m-file script that tests your file.

Its name should be Voltage_divider_test_Name.m, where you replace "Name" with your name.

This file sets each of these sets of inputs and runs (calls) your file:

a:

```
Vin = 5 volts;
R1 = 2000 Ohms;
R2 = 8000 Ohms
Voltage_divider_Name  % This line runs (calls) your script.
```

b.

```
Vin = 10 volts;
R1 = 5000 Ohms;
R2 = 5000 Ohms
Voltage_divider_Name  % This line runs (calls) your script.
```

Answer

Write the code code and test it.

.

This page titled 2.17.1: More Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

3: for Loops, Ploting Points, Sequences, and Series

3.1: Updating Variables
3.2: Bug Taxonomy
3.3: Absolute and Relative Error
3.4: for Loops
  3.4.1: for Loop Supplemental Material
3.5: Plotting Points
  3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop
3.6: Sequences
  3.6.1: Sequences Additional Material
3.7: Series
  3.7.1: Series Additional Material
3.8: Generalization
3.9: Incremental Development
3.10: for Loop Incremental Development Exercise
3.11: Chapter Review
3.12: Chapter 3 for loop Videos in English and Espanol
3.13: Exercises

Most of the material in this chapter is from Alan B. Downey, "Physical Modeling in Matlab", chapter 3.

Additional material has been added by Carey A Smith.

This page titled 3: for Loops, Ploting Points, Sequences, and Series is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

3.1: Updating Variables

In Exercise 2.2 of Section 2.9, I asked you to write a program that models a bike-share system with bikes moving between two stations. Each day 5 percent of the bikes in Boston are dropped off in Cambridge, and 3 percent of the bikes in Cambridge get dropped off in Boston.

To update the state of the system, you might have been tempted to write something like

```
b = b - 0.05*b + 0.03*c
c = c + 0.05*b - 0.03*c
```

But that would be wrong, so very wrong. Why? The problem is that the first line changes the value of `b`, so when the second line runs, it gets the old value of `c` and the new value of `b`. As a result, the change in `b` is not always the same as the change in `c`, which violates the Principle of Conservation of Bikes!

One solution is to use temporary variables like `b_new` and `c_new`:

```
b_new = b - 0.05*b + 0.03*c
c_new = c + 0.05*b - 0.03*c
b = b_new
c = c_new
```

This has the effect of updating the variables *simultaneously*; that is, it reads both old values before writing either new value.

The following is an alternative solution that has the added advantage of simplifying the computation:

```
b_to_c = 0.05*b - 0.03*c
b = b - b_to_c
c = c + b_to_c
```

It's easy to look at this code and confirm that it obeys Conservation of Bikes. Even if the value of `b_to_c` is wrong, at least the total number of bikes is right. And that brings us to the Fifth Theorem of Debugging:

> *The best way to avoid a bug is to make it impossible.*

In this case, removing redundancy also eliminates the opportunity for a bug.

This page titled 3.1: Updating Variables is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.1: Updating Variables** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.2: Bug Taxonomy

The more you understand bugs, the better you will be at debugging. There are four kinds of bugs:

Syntax error

> You have written a command that cannot execute because it violates one of the language's syntax rules. For example, in MATLAB, you can't have two operands in a row without an operator, so `pi r^2` contains a syntax error. When the interpreter finds a syntax error, it prints an error message and stops running your program.

Runtime error

> Your program starts running, but something goes wrong along the way. For example, if you try to access a variable that doesn't exist, that's a runtime error. When the interpreter detects the problem, it prints an error message and stops.

Logical error

> Your program runs without generating any error messages, but it doesn't do the right thing. The problem in the previous section, where we changed the value of `b` before reading the old value, is a logical error.

Numerical error

> Most computations in MATLAB are only approximately right. Most of the time the errors are small enough that we don't care, but in some cases the round-off errors are a problem.

Syntax errors are usually the easiest to deal with. Sometimes the error messages are confusing, but MATLAB can usually tell you where the error is, at least roughly.

Runtime errors are harder because, as I mentioned before, MATLAB can tell you where it detected the problem, but not what caused it.

Logical errors are hard because MATLAB can't help at all. From MATLAB's point of view there's nothing wrong with the program; only you know what the program is supposed to do, so only you can check it.

Numerical errors can be tricky because it's not clear whether the problem is your fault. For most simple computations, MATLAB produces the floating-point value that is closest to the exact solution, which means that the first 15 significant digits should be correct.

But some computations are ill-conditioned, which means that even if your program is correct, the round-off errors accumulate and the number of correct digits can be smaller. Sometimes MATLAB can warn you that this is happening, but not always! *Precision* (the number of digits in the answer) does not imply *accuracy* (the number of digits that are right).

This page titled 3.2: Bug Taxonomy is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.2: Bug Taxonomy** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.3: Absolute and Relative Error

There are two ways of thinking about numerical errors. The first is *absolute error*, or the difference between the correct value and the approximation. We often write the magnitude of the error, ignoring its sign, when it doesn't matter whether the approximation is too high or too low.

The second way to think about numerical errors is *relative error*, where the error is expressed as a fraction (or percentage) of the exact value.

For example, we might want to estimate $9!$ using the formula $\sqrt{18\pi}(9/e)^9$. The exact answer is $9\cdot 8\cdot 7\cdot 6\cdot 5\cdot 4\cdot 3\cdot 2\cdot 1 = 362,880$. The approximation is $359,536.87$. So the absolute error is $3,343.13$.

At first glance, that sounds like a lot—we're off by three thousand—but we should consider the size of the thing we are estimating. For example, $3,000 matters a lot if we're talking about an annual salary, but not at all if we're talking about the national debt.

A natural way to handle this problem is to use relative error. In this case, we would divide the error by $362,880$, yielding 0.00921, which is just less than 1 percent. For many purposes, being off by 1 percent is good enough.

This page titled 3.3: Absolute and Relative Error is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.3: Absolute and Relative Error** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.4: for Loops

Let's return to the bike-share example. In the previous chapter, we wrote a *bikeupdate.m* script to simulate a day in the life of a bike-share system. To simulate an entire month, we'd have to run the script 30 times. We could enter the same command 30 times, but it's simpler to use a *loop*, which is a set of statements that executes repeatedly.

To create a loop, we can use the `for` statement, like this:

```
for i=1:30
    bike_update
end
```

The first line includes what looks like an assignment statement, and it *is* like an assignment statement, except that it runs more than once. The first time, it creates the variable `i` and assigns it the value 1. The second time, `i` gets the value 2, and so on, up to and including 30.

The colon operator (`:`) specifies a *range* of integers. You can create a range at the prompt:

```
>> 1:5
ans =   1     2     3     4     5
```

The variable you use in the `for` statement is called the *loop variable*. It's common to use the names `i` , `j` , and `k` as loop variables.

The statements inside the loop are called the *body*. By convention, they are indented to show that they're inside the loop, but the indentation doesn't affect the execution of the program. The `end` statement marks the end of the loop.

To see a loop in action you can run one that displays the loop variable:

```
>> for i=1:5
    i
end

i = 1
i = 2
i = 3
i = 4
i = 5
```

As this example shows, you *can* run a `for` loop from the command line, but it's more common to put it in a script.

? Exercise 3.1

Create a script named *bikeloop.m* that uses a `for` loop to run *bikeupdate.m* 30 times. Before you run it, you have to assign values to `b` and `c` . For this exercise, start with the values `b = 100` and `c = 100` .

If everything goes smoothly, your script will display a long stream of numbers on the screen. It's probably too long to fit, and even if it did fit, it would be hard to interpret. A graph would be much better!

This page titled 3.4: for Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.4: for Loops** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.4.1: for Loop Supplemental Material

By Carey A. Smith

3.4.1.1: Structure of a for loop

Every for loop must start with the key word "for" followed by the loop index and its values.

Then the body of the for loop is whatever code your want.

Then the keyword "end" specifies the end of the loop code.

This is a simple example:

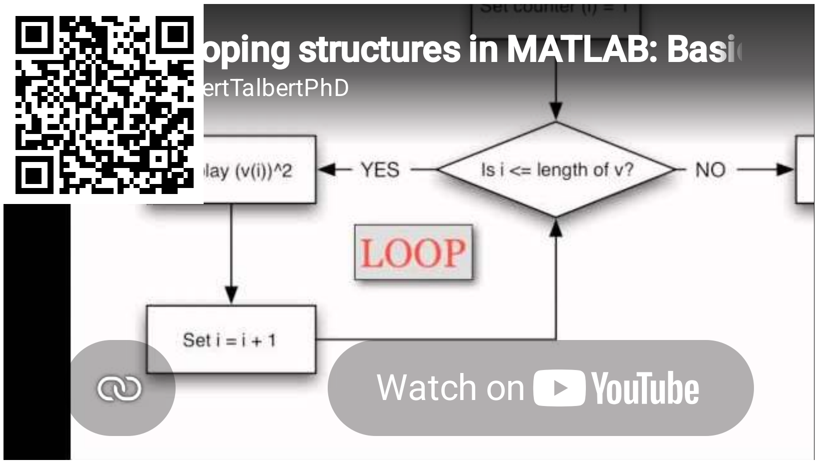
```
for kk = 1:4

     x = cos(kk*pi/16);

     y = sin(kk*pi/16);

end
```

For good style, vertically align the "for" and "end". Indent the body of the for loop.

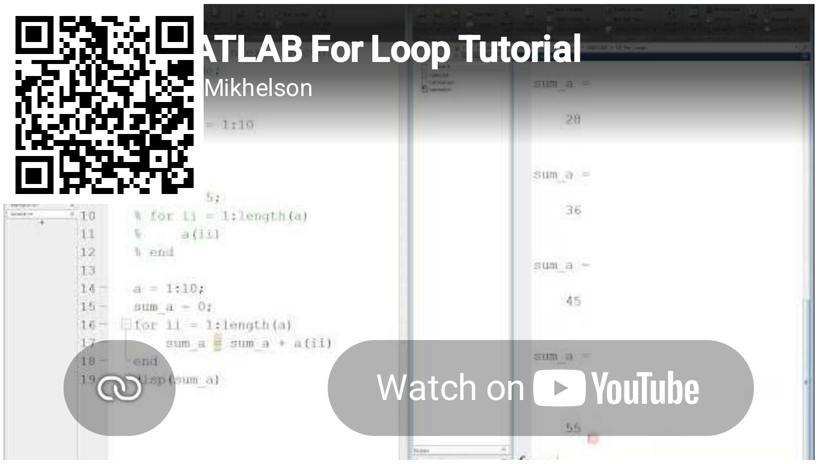
You may use the index (kk in this example) in the calculations, as shown in this example. But never modify the loop index inside the for loop. MATLAB will increment the index automatically.

3.4.1.2: Watch these videos for additional instruction on for loops:

Video: Looping structures in MATLAB: Basic FOR loops, Robert Talbert


oping structures in MATLAB: Basi
ertTalbertPhD
lay (v(i))^2
YES
Is i <= length of v?
NO
LOOP
Set i = i + 1
Watch on YouTube

Video: MATLAB For Loop Tutorial, Ilya Mikhelson


ATLAB For Loop Tutorial
Mikhelson
Watch on YouTube

```
% for ii = 1:length(a)
%     a(ii)
% end

a = 1:10;
sum_a = 0;
for ii = 1:length(a)
    sum_a = sum_a + a(ii)
end
disp(sum_a)
```

```
sum_a =
   28
sum_a =
   36
sum_a =
   45
sum_a =
   55
```

3.4.1.3: Modify Bike_update to round the number of bikes at the end of the day

The original version of Bike_update.m allows a fraction of a bike to get transferred, which is unrealistic. This modified version rounds the number of bikes, since only whole bikes are transferred.

✓ Example 3.4.1.1 Bike_loop with whole number of bikes

The number of bikes transferred is rounded to hole numbers with these lines of code:

```
b2c = round(b1*0.05) % Compute the number of bikes that move from B to C
c2b = round(c1*0.03) % Compute the number of bikes that move from C to B
```

Solution

Add example text here.

? Exercise 3.4.1.1 Modified Bike Update for loop with plot

Create a script named "bike_loop_YourName.m" that uses a for loop to run bike_update.m 30 times. This simulates what would happen during 1 month.

```
%% Bike_update (Exercise 2.2)
% This script models a bike-share system
%  with bikes in Boston and Cambridge, Massachusetts
%
% Preconditions (Inputs):
%  Values set before this script is called:
% b = the number of bikes in Boston (B) at the beginning of the day
% c = the number of bikes in Cambridge (C) at the end of the day
%
% Postconditions (Outputs):
%  The values of b and c at the end of the day
%
% Algorithm: every day:
%  5 percent (0.05) of the bikes in Boston are dropped off in Cambridge,
%  3 percent (0.03) of the bikes in Cambridge get dropped off in Boston.
% Use the round function to compute the number of bikes that move each day.
%
%% Save the input values, so that they don't change while
%  computing the updated values
b1 = b; % Save the input value
c1 = c; % Save the input value
% Compute the numbers of bikes that move
b2c = round(b1*0.05) % Compute the number of bikes that move from B to C
c2b = round(c1*0.03) % Compute the number of bikes that move from C to B
%% Resulting values:
b = b1 - b2c + c2b % initial value - those that leave
%                                  + those that arrive
c = c1 - c2b + b2c % initial value - those that leave
%                                  + those that arrive
```

Write a script (m-file) that uses a for loop to run bike_update.m 30 times.

1. Start by clearing any prior variables and scripts:

```
clear all; close all; format compact;
```

Set b and c to these values:

```
b = 100; % Boston, MA
```

```
c = 100; % Cambridge, MA
```

2. Open a new figure and plot the numbers of bikes in each location on the first day with this code:

```
figure;
```

```
hold on; % This allows us to plot multiple points on one figure.
```

```
plot(0, b, 'b*')  % Plot the number of bikes in Boston at the start with a blue asterisk.
```

```
plot(0, c, 'ro')  % Plot the number of bikes in Cambridge at the start with a red circle.
```

```
grid on;
```

% 3. Then put this "for loop" in your code:

```
for k = 1:30
    bike_update
end
```

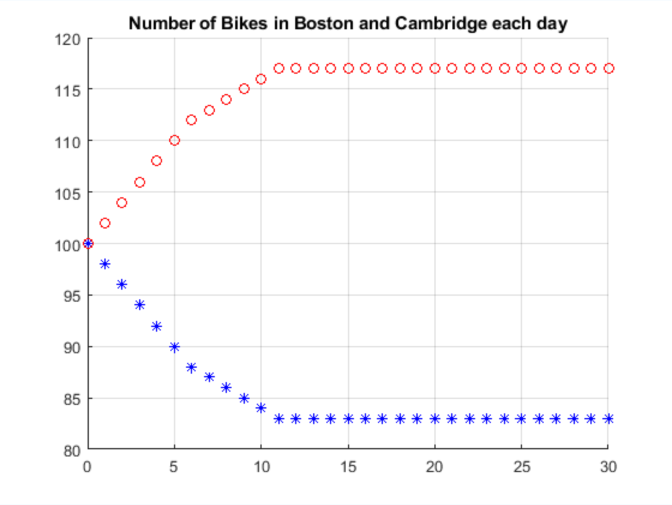
% Inside the for loop, add this code to plot each day's result:

```
    plot(k, b, 'b*')  % Plot the number of bikes in Boston on day k
    plot(k, c, 'ro')  % Plot the number of bikes in Cambridge on day k
```

% Add a title to the figure with this code.

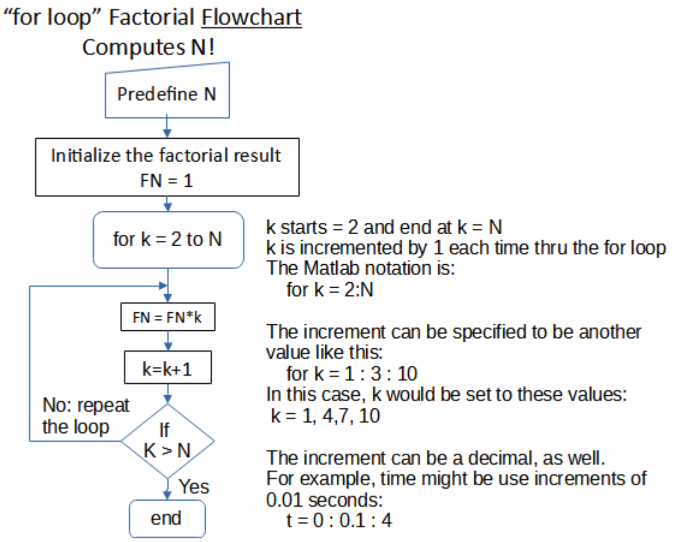
```
title('Number of Bikes in Boston and Cambridge each day')
```

Answer



Add texts here. Do not delete this text first.

3.4.1.4: Here are 2 more examples of for loops:


“for loop” Factorial Flowchart
Computes N!
Predefine N
Initialize the factorial result
FN = 1
for k = 2 to N
FN = FN*k
k=k+1
No: repeat the loop
If K > N
Yes
end

k starts = 2 and end at k = N
k is incremented by 1 each time thru the for loop
The Matlab notation is:
for k = 2:N

The increment can be specified to be another value like this:
for k = 1 : 3 : 10
In this case, k would be set to these values:
k = 1, 4,7, 10

The increment can be a decimal, as well.
For example, time might be use increments of 0.01 seconds:
t = 0 : 0.1 : 4

```
Figure \(\PageIndex{i}\): Factorial for loop flowchart
```

✓ Example 3.4.1.1 Factorial Example for loop

The factorial of a number N in algebra is defined as:

N! = 1*2*3*...*N

This can be computed in MATLAB/Octave using a for loop using the logic described in accompanying figure.

Solution

The MATLAB code for this is:

```
% Input (precondition): N = integer >= 2
% Output: FN
FN = 1; % Starting value
for k = 2:N % k=index, kstart : to kend
   FN = FN*k
end
```

? Exercise 3.4.1.1 Compute x^n using a for loop

Compute x^n these 2 ways:

1a. Set x = 2

1b. Set n = 7

1c. Compute a = x^n.

2a. Set result = 1

2b. Code a for loop that multiplies result by x, n times.

2c. Use this line inside the for loop:

```
result = result*x
```

Answer

```
x = 2 % Initialze x
n = 7 % Initialze n (the power)
result = 1 % Initialze the result
for k=1:n
   result = result*x
end
```

.

This page titled 3.4.1: for Loop Supplemental Material is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

3.5: Plotting Points

If the output of your program is a long stream of numbers, it can be hard to see what is happening. Plotting the results can make things clearer.

The `plot` function is a versatile tool for plotting two-dimensional graphs. Unfortunately, it's so versatile that it can be hard to use (and hard to read the documentation). We'll start simple and work our way up.

3.5.1: Graphics Toolkit for Octave

If you are using Octave, you need to activate a graphics toolkit before creating a plot. Usually one of these 2 options will work, depending on your computer's graphics and on your version of Octave:

```
graphics_toolkit("qt")
graphics_toolkit("fltk")
```

Use only "qt" or "fltk", not both. More information can be found at:

https://docs.octave.org/v6.1.0/Introduction-to-Plotting.html

The rest of this section applies to both MATLAB and Octave.

3.5.2: Plot a single point with figure() and plot()

Use the `figure` function to create a figure window.

To plot a single point, type

```
>> plot(1, 2, 'o')
```

A Figure Window should appear with a graph and a single blue circle at x position 1 and y position 2.

The letter in single quotes is a *style string* that specifies how the point should be plotted; `o` indicates a circle. Other shapes include `+`, `*`, `x`, `s` (for a square), `d` (for a diamond), and `^` (for a triangle).

You can also specify the color by starting the style string with a color code:

```
>> plot(1, 2, 'ro')
```

Here, `r` stands for red; the other colors include `g` for green, `b` for blue, `c` for cyan, `m` for magenta, `y` for yellow, and `k` for black.

When you use `plot` this way, it can only plot one point at a time. If you run `plot` again, it clears the figure before making the new plot. The `hold` command lets you override that behavior: `hold on` tells MATLAB not to clear the figure when it makes a new plot; `hold off` returns to the default behavior.

Try this:

```
>> clf
>> hold on
>> plot(1, 1, 'ro')
>> plot(2, 2, 'go')
>> plot(3, 3, 'bo')
>> hold off
```

The `clf` command clears the figure before we start plotting.

If you run the code above, you should see a figure with three circles. MATLAB scales the plot automatically so that the axes run from the lowest values in the plot to the highest.

Exercise 3.2

Modify *bikeloop.m* so that it clears the figure before running the loop. Then, each time through the loop, it should plot the value of `b` versus the value of `i` with a red circle.

Once you get that working, modify it so it plots the values of `c` with blue diamonds.

This page titled 3.5: Plotting Points is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.5: Plotting** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop

- By Carey A. Smith

Page ID 84378

2-Dimensional plots of a time sequence can be easily created in MATLAB and in Octave.

Octave needs a graphics toolkit to plot graph. MATLAB does not need this.

Insert this line in your code before creating your first figure:

```
graphics_toolkit("qt") % This is for Octave only. Do not use it with MATLAB.
```

Also, the Octave fltk toolkit is obsolete. Do not use the fltk toolkit.
`graphics_toolkit("fltk")` For more information on graphics toolkit options, see:

https://docs.octave.org/interpreter/Graphics-Toolkits.html

Again, the graphics toolkit is only for Octave. It is not needed for MATLAB and will cause an error if attempted.

The key lines of code for creating a plot in both MATLAB and Octave are:

- `figure; % Open a figure window. The plot is created in this figure window.`
- `plot(t, sin_t, 'o')`
  - The first variable, t, is the x coordinate of the point.
  - The second variable, sin_t, is the y coordinate of the point.
  - The 'o' means to use a circle for point's marker.
- `hold on; % Let additional points be plotted on the same figure`

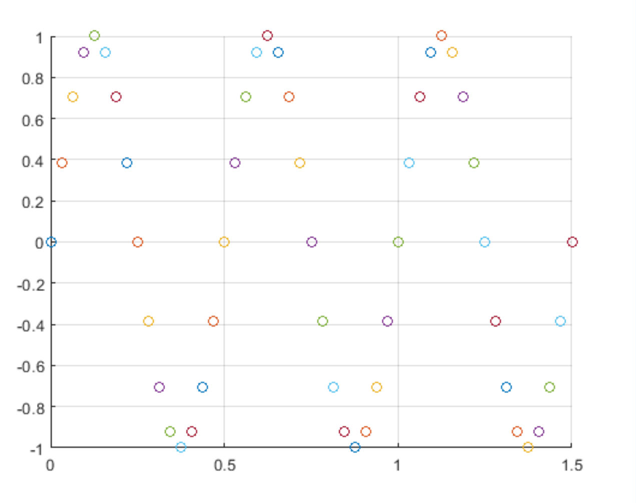
It can be useful to code this example together as a class.

✓ Example 3.5.1.1 Plot points of a sine wave

```
% Compute and plot points of a sine wave
% using a script with a "for loop"
clear all; % Clear all variables
clc; % Clear the console (computer screen)
close all; % close all figures
format compact; % Don't insert blank lines on the console

graphics_toolkit("fltk") % Include this line when using Octave. Not for Matlab.

%% Initialize variables
period = 0.5; % seconds
t_end = 3*period; % The end of the interval
dt = period/16; % time between points
```

```
%%
figure; % Open a figure window
xlim([0,t_end]) % Set the x-axis to be 0 to t_end
ylim([-1,1]) % Set the x-axis to be -1 to 1
grid on; % Draw grid lines
hold on; % Let additional points be plotted on the same figure
%%
for t = 0 : dt : t_end % t = 0, dt, 2*dt, ... t_end
    % Compute the sine for the current value of t
    sin_t = sin(2*pi*t/period);
    plot(t, sin_t, 'o')
    pause(0.3) % pause long enuf to see each point as it is plotted
end
```

Solution



This page titled 3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

3.6: Sequences

Now that we have the ability to write loops, we can use them to explore sequences and series, which are useful for describing and analyzing systems that change over time.

In mathematics, a *sequence* is a set of numbers that corresponds to the positive integers. The numbers in the sequence are called *elements*. In math notation, the elements are denoted with subscripts, so the first element of the series A is A_1, followed by A_2, and so on.

A `for` loop is a natural way to compute the elements of a sequence. As an example, in a geometric sequence, each element is a constant multiple of the previous element. As a more specific example, let's look at the sequence with $A_1 = 1$ and the relationship $A_{i+1} = A_i/2$, for all i. In other words, each element is half as big as the one before it.

The following loop computes the first 10 elements of A:

```
a = 1
for i=2:10
    a = a/2
end
```

The first line initializes the variable `a` with the first element of the sequence, A_1. Each time through the loop, we find the next value of `a` by dividing the previous value by 2, and assign the result back to `a`. At the end, `a` contains the 10th element.

The other elements are displayed on the screen, but they are not saved in a variable. Later, we'll see how to save the elements of a sequence in a vector.

This loop computes the sequence *recurrently*, which means that each element depends on the previous one. For this sequence, it's also possible to compute the ith element *directly*, as a function of i, without using the previous element.

In math notation, $A_i = A_1 r^{i-1}$, where r is the ratio of successive elements. In the previous example, $A_{i+1} = A_i/2$, so $r = 1/2$.

? Exercise 3.3

Write a script named *sequence.m* that uses a loop to compute elements of A .

This page titled 3.6: Sequences is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.6: Sequences** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.6.1: Sequences Additional Material

By Carey A. Smith

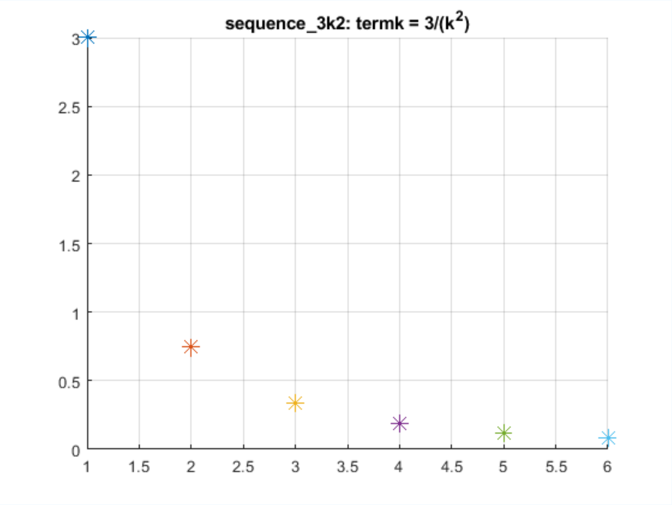
✓ Example 3.6.1.1 Sequence 3/(k^2)

This example computes a sequence with terms = 3/(k^2)

```
clear all; close all; format compact; clc;
```

```
% Octave needs graphics toolkit to plot graph.
graphics_toolkit("fltk") % Do not use with MATLAB
figure; % Open a figure
hold on; % Plot all points on this figure
grid on;
n = 6 % The number of terms in the sequence
for k=1:n
    termk = 3/(k^2)
    plot(k,termk,'*','MarkerSize',10)
end
title('sequence\_3k2: termk = 3/(k^2)')
```

Solution



? Exercise 3.6.1.1 Fibonacci #2

Leonardo Fibonacci was an Italian mathematician, who lived around 1200 AD. He traveled to Arab countries and learned about Arab and Hindu progress in math and algebra. He popularized the Hindu–Arabic numeral system in the Western world.Hindu–Arabic numeral system(opens in new window) [en.Wikipedia.org] It is much better for arithmetic and algebra than Roman numerals. This enabled the math and science advances of the Renaissance. He is most famous for his "Fibonacci sequence".

The first terms in Fibonacci sequence are:

0, 1, 1, 2, 3, 5, 8, 13, ...

Each term successive is the sum of the 2 preceding terms.

Examples:

2+3 = 5

3+5 = 8

(1 pt) Create a MATLAB/Octave m-file script named "Fibonacci_2_YourName.m" (Replace YourName with your own name.)

(1 pt) Put these lines of code at the beginning of your file:

```
% Compute a Fibonacci sequence with a for loop
% Clear any variables; close any figures; eliminate white space; clear the console:
clear all; close all; format compact; clc;
```

```
% Octave needs graphics toolkit to plot graph.
graphics_toolkit("fltk") % Do not use with MATLAB
figure; % Open a figure window.
hold on; % Allow multiple points to be plotted on the same figure.
grid on; % Draw horizontal and vertical grid lines on the plot.
```

(2 pts) Initialize and plot the 1st 2 terms with this code:

```
a = 0 % 1st number in the sequence
plot(1,a,'bo')
b = 1 % 2nd number in the sequence
plot(2,b,'bo')
```

(2 pts) Write a for loop. Let k = the for-loop iteration number. k goes from 3 to 12

(2 pts) Inside the for loop, calculate and plot the next term with this code:

```
c = a+b
plot(k,c,'bo')
```

(2 pts) Then update a and b to be the current last 2 terms, in order to be ready for the next iteration. This is similar to the bike example.

```
a = b; % Update a for the next iteration. (Use a semicolon so that this line does not display.)
b = c; % Update a for the next iteration. (Use a semicolon so that this line does not display.)
```

Answer

Add texts here. Do not delete this text first.

This page titled 3.6.1: Sequences Additional Material is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

3.7: Series

In mathematics, a *series* is the sum of the elements of a sequence. It's a terrible name, because in common English, "sequence" and "series" mean pretty much the same thing, but in math, a sequence is a set of numbers, and a series is an expression (a sum) that has a single value. In math notation, a series is often written using the summation symbol $\sum$.

For example, the sum of the first 10 elements of A is

$$\sum_{i=1}^{10} A_i$$

A `for` loop is a natural way to compute the value of this series:

Listing 3.1: A program that calculates a simple series

```
A1 = 1;
total = 0;
for i=1:10
    a = A1 * (1/2)^(i-1);
    total = total + a;
end
ans = total
```

Let's walk through what's happening here. `A1` is the first element of the sequence, so we assign it to be `1`; we also create `total`, which will store the cumulative sum. Each time through the loop, we set `a` to the ith element and add `a` to the `total`. At the end, outside the loop, we store `total` as `ans`

The way we're using `total` is called an *accumulator*, that is, a variable that accumulates the result a little bit at a time.

Exercise 3.4

This example computes the terms of the series directly. As an exercise, write a script named *series.m* that computes the same sum by computing the elements recurrently. You will have to be careful about where you start and stop the loop.

This page titled 3.7: Series is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.7: Series** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.7.1: Series Additional Material

By Carey A. Smith

There are 2 methods to sum a series. Let the variable "term" be computed value of the current value and the variable "total" be the running sum.

Method 1: Use a temporary variable, "total_new", to make it clear what is happening:

`total_new = total + term;` % Add the previous total and the current term to get the new total.

`total = total_new;` % Reset total to the new value

Method 2: Directly assign the new value of "total" to be equal to current value of total + term:

`total = total + term;` % Replace the previous total with the sum of the current term + term

These 2 methods are equivalent. The first method is more obvious what is happening. The second is method common among experienced programmers. It may be slightly faster.

Note: Variable names other than "total"--such as "series_sum--are often used in codes.

Warning: Do not using "sum" as a variable, because that would be a name collision with the built-in function sum().

.

✓ Example 3.7.1.1 Power Series1

```
% Clear any variables etc.
clear all; close all; format compact; clc;
% Initialize these variables:
n = 6;    % The number of terms in the series
A0 = 4; % The first value
r = 1/2;  % The ratio of successive terms
% Write a for loop that computes the terms in the series which are computed with
this expression:
%   A(k) = A0*r^k;

%% Method 1:
total = 0;  % Initialize the total
for k = 1:n
  A(k) = A0*r^k;
  total_new = total + A(k);
  total = total_new;
end
A % This displays all the values of the vector A
disp(['Method 1 total = ',num2str(total)])

%% Method 2:
total = 0;  % Initialize the total
for k = 1:n
  A(k) = A0*r^k;
  total = total + A(k);
end
A % This displays all the values of the vector A
disp(['Method 2 total = ',num2str(total)])
```

Solution

Add example text here.

.

Exercise 3.7.1.1 atan(x) Taylor's series and absolute error

The Taylor's series for the arctangent function is:

$\text{atan}(x) = x - x^3/3 + x^5/5 - x^7/7 + \ldots$

The general formula for the k^{th} term is computed with these 2 lines of code:

```
m = (2*k-1) % This generates 1, 3, 5, 7, ...
```

```
term = (-1)^(k-1)*x^m / m   % (-1)^(k-1) = 1, -1, 1, -1, ...
```

Because the sign changes from term to term, this is called an alternating series.

(1 pt) Write a Matlab m-file script. This script will have a "for loop". The details of the for loop are described below.

(1 pt) Put these lines of code at the beginning of your file:

```
% Compute the Taylor's series for atan(x)
% Clear any variables; close any figures; eliminate white space; clear the console
clear all; close all; format compact; clc;
% Open a figure for plotting the partial sums
```

```
% Octave needs graphics toolkit to plot graph.
graphics_toolkit("fltk") % Do not use with MATLAB
figure;
hold on;
grid on;
```

(1 pt) Initialize these variables:

```
x = pi/5 % Set the x-value
atan_series = 0; % Initialize the series' sum
```

(2 pts) Write a for loop. Let k = the for-loop index. k goes from 1 to 8

(3 pts) Inside the for loop, compute term as specified above for each iteration.

Then add term to the previous value of atan_series with this code (method 1):

```
atan_series_new = atan_series + term
```

```
atan_series = atan_series_new
```

(1 pt) Inside the for loop, also plot the kth partial sum with this code:

```
plot(k, atan_series,'o');
```

(1 pt) After the for loop, display the sum and the last term with these lines of code:

```
atan_series % Display the series sum
```

```
term % Display the last term that was calculated
```

(1 pt) Compute and display atan_Matlab = atan(x) [Matlab's buit-in function]

(1 pt) Compute and display the absolute error = abs(atan_series - atan_Matlab)

If the code is done correctly, the absolute error should be < 0.001

Answer

Add texts here. Do not delete this text first

.

Example 3.7.1.1 Bessel Function

The image of a star thru a space telescope, which has no aberrations, is called an Airy pattern, shown in this image:

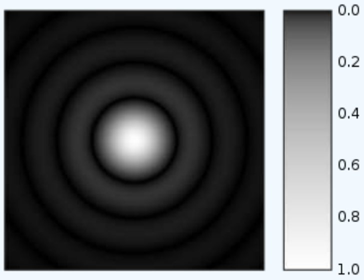


Figure 3.7.1.1: Airy pattern

(Airy pattern image [en.Wikipedia.org], This work has been released into the public domain by its author, Sakurambo at English Wikipedia.)

The formula for a cross-section of the Airy pattern intensity is:

$$I(\theta)=I_0\left[\frac{2J_1(k*a*sin\theta)}{k*a*sin\theta}\right]^2=I_0\left[\frac{2J_1(x)}{x}\right]^2 \tag{3.7.1.1}$$

where J_1(x) is the 1st-order **Bessel function of the 1st kind**.

This Bessel function is approximated by this infinite series with α = 1:

$$J_\alpha(x)=\sum_{m=0}^{\infty}\frac{(-1)^m}{m!*\Gamma(m+\alpha+1)}*\left(\frac{x}{2}\right)^{(2m+\alpha)} \tag{3.7.1.2}$$

For m!, use the Matlab function factorial(m)

Γ(p) = the Gamma function. In Matlab this function is gamma(p).

When is a positive integer, gamma(p) = (p-1)!

Instructions:

a. Set x = 0.5
b. Write a for loop to sum the terms of this Bessel function series, J_α(x), for m = 0 to 10.
c. Display the last term
d. Display the resulting sum

Solution

term = 1.5697e-27
J1_x = 0.24227

..

This page titled 3.7.1: Series Additional Material is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

3.8: Generalization

As written, the previous example always adds up the first 10 elements of the sequence, but we might be curious to know what happens to `total` as we increase the number of terms in the series. If you've studied geometric series, you might know that this series converges on 2; that is, as the number of terms goes to infinity, the sum approaches 2 asymptotically.

To see if that's true for our program, we can replace the constant `10` in Listing 3.1 with a variable named `n` :

Listing 3.2: Updating our code from Listing 3.1 to have a variable number of terms

```
A1 = 1;
total = 0;
for i=1:n
    a = A1 * (1/2)^(i-1);
    total = total + a;
end
ans = total
```

The code in Listing 3.2 can now compute any number of terms, with the precondition that you have to set `n` before you execute the code. I put this code in a file named *series.m*, then ran it with different values of `n` :

```
>> n=10; series
total = 1.99804687500000

>> n=20; series
total = 1.99999809265137

>> n=30; series
total = 1.99999999813735

>> n=40; series
total = 1.99999999999818
```

It sure looks like it's converging on 2.

Replacing a constant with a variable is called *generalization*. Instead of computing a fixed, specific number of terms, the new script is more general; it can compute any number of terms. This is an important idea we'll come back to when we talk about functions.

This page titled 3.8: Generalization is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.8: Generalization** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.9: Incremental Development

As you start writing longer programs, you might find yourself spending more time debugging. The more code you write before you start debugging, the harder it is to find the problem.

Incremental development is a way of programming that tries to minimize the pain of debugging by developing and testing in small steps. The fundamental steps are:

1. Always start with a working program. If you have an example from a book, or a program you wrote that is similar to what you are working on, start with that. Otherwise, start with something you *know* is correct, like `x = 5`. Run the program and confirm that you are running the program you think you are running. This step is important, because in most environments there are little things that can trip you up when you start a new project. Get them out of the way so you can focus on programming.
2. Make one small, testable change at a time. A *testable* change is one that displays something on the screen (or has some other effect) that you can check. Ideally, you should know what the correct answer is or be able to check it by performing another computation.
3. Run the program and see if the change worked. If so, go back to step 2. If not, you'll have to do some debugging, but if the change you made was small, it shouldn't take long to find the problem.

With incremental development, your code is more likely to work the first time, and if it doesn't, the problem is more likely to be obvious. And that brings us to the Sixth Theorem of Debugging:

> *The best kind of debugging is the kind you don't have to do.*

In practice, there are two problems with incremental development. First, sometimes you have to write extra code to generate visible output that you can check. This extra code is called *scaffolding* because you use it to build the program and then remove it when you are done. But the time you save on debugging is almost always worth the time you invest in scaffolding.

The second problem is that when you're getting started, you might not know how to choose steps that get from `x = 5` to the program you're trying to write. We'll look at an extended example in Chapter 6.

If you find yourself writing more than a few lines of code before you start testing and you're spending a lot of time debugging, you should try incremental development.

This page titled 3.9: Incremental Development is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.9: Incremental Development** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.10: for Loop Incremental Development Exercise

? Exercise 3.10.1 for Loop Incremental Development Exercise

Write a for loop that does the following.

Start with any existing for loop, preferably one that sums a series, such as the one in section 3.8.

Test you code after you code each step in the process.

1. Opens a figure for plotting.
2. Sets the number of terms in the series to be n = 20
3. Initializes series_sum = 0; % (Do not name the variable "sum", becuz that is a built-in MATLAB function.)
4. Creates a for loop with index k = 1:n
5. Computes termk = 3/(k^2)
6. Adds termk to series_sum, and stores the result back into the series_sum variable.
7. For each value of k, plots termk with the '*' marker.
8. For each value of k, plots series_sum with the 's' marker.
9. After the for loop, display the final values of termk and series_sum.
10. Adds a title to the plot.

Answer

Add texts here. Do not delete this text first.

This page titled 3.10: for Loop Incremental Development Exercise is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

3.11: Chapter Review

In this chapter, we used a loop to perform a repetitive computation—updating a model 30 times—and to compute sequences and series. Also, we used the `plot` function to visualize the results.

Here are some terms from this chapter you might want to remember.

Absolute error is the difference between an approximation and an exact answer. *Relative error* is the same difference expressed as a fraction or percentage of the exact answer.

A *loop* is part of a program that runs repeatedly. A *loop variable* is a variable that gets assigned a different value each time through the loop. A *range* is a sequence of values assigned to the loop variable, often specified with the colon operator—for example, `1:5` . The *body* of a loop is the set of statements inside the loop that runs repeatedly. An *accumulator* is a variable that is used to accumulate a result a little bit at a time.

In mathematics, a *sequence* is a set of numbers that correspond to the positive integers. The numbers that make up the sequence are called . A *series* is the sum of a sequence of elements. Sometimes we compute the elements of a sequence *recurrently*, which means that each new element depends on previous elements. Sometimes we can compute the elements *directly*, without using previous elements.

Generalization is a way to make a program more versatile, for example, by replacing a specific value with a variable that can have any value. *Incremental development* is a way of programming by making a series of small, testable changes. *Scaffolding* is code you write to help you program or debug but that is not part of the finished program.

In the next chapter, we'll use a vector to store the results from a loop.

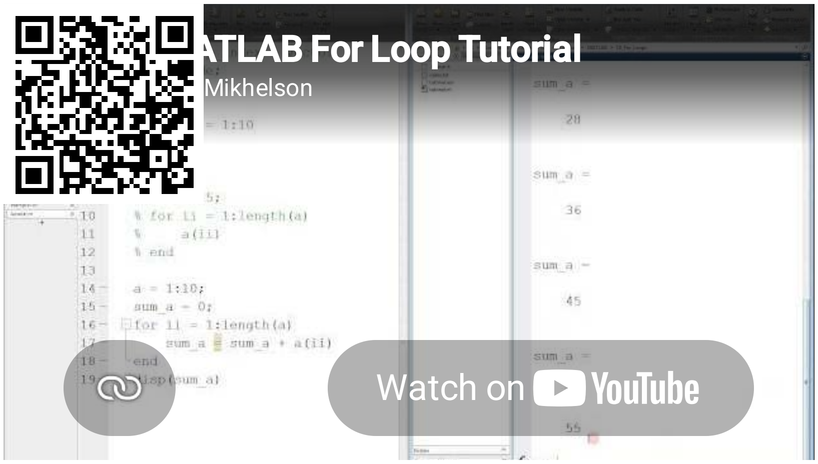
This page titled 3.11: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.10: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

3.12: Chapter 3 for loop Videos in English and Espanol

English

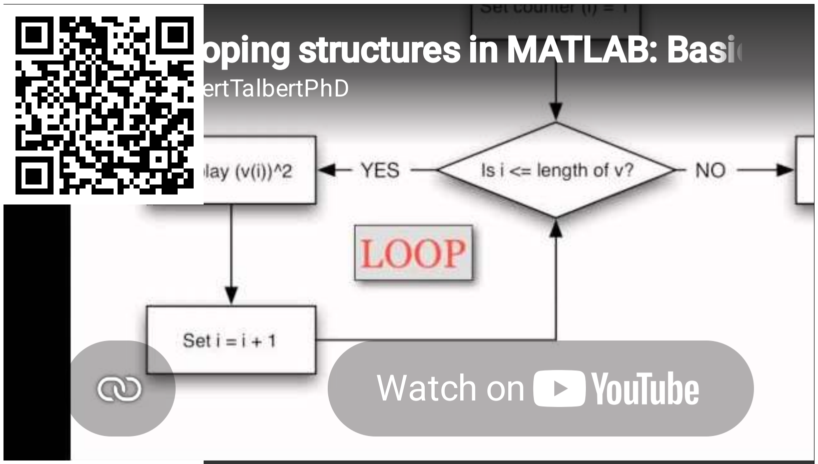
MATLAB For Loop Tutorial (Ilya Mikhelson)

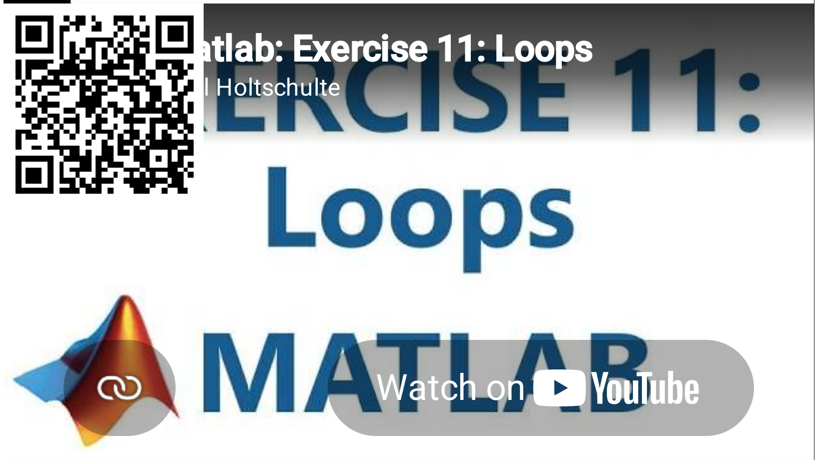

ATLAB For Loop Tutorial
Mikhelson
Watch on YouTube

```
% for ii = 1:length(a)
%     a(ii)
% end

a = 1:10;
sum_a = 0;
for ii = 1:length(a)
    sum_a = sum_a + a(ii)
end
disp(sum_a)
```

```
sum_a =

    28

sum_a =

    36

sum_a =

    45

sum_a =

    55
```

Basic FOR loops with if-else, RTalbert


oping structures in MATLAB: Basi
ertTalbertPhD
Set counter (i) = 1
lay (v(i))^2
YES
Is i <= length of v?
NO
LOOP
Set i = i + 1
Watch on YouTube


atlab: Exercise 11: Loops
l Holtschulte
ERCISE 11:
Loops
MATLAB
Watch on YouTube

Matlab While Loop video, RTalbert

oping structures in MATLAB: WHI
ertTalbertPhD
Watch on YouTube

Matlab Loop types, while, for, nested (Univ. of Wyoming) (PDF file)

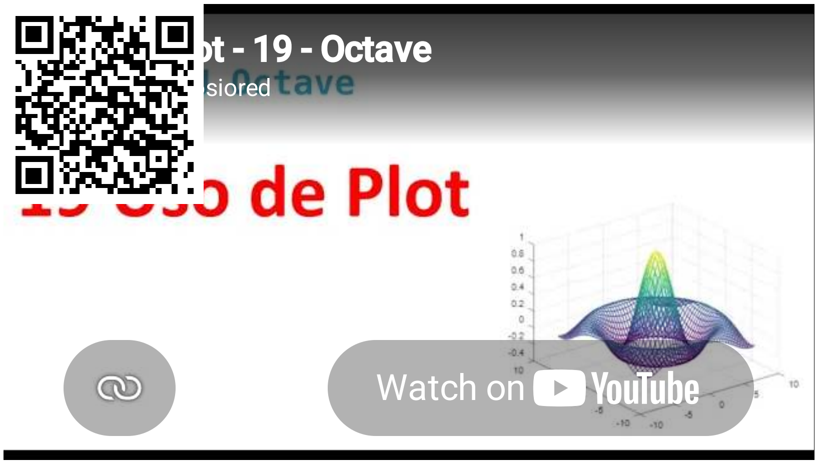
Matlab Loop types, while, for, nested [www.uwyo.edu]

Espanol

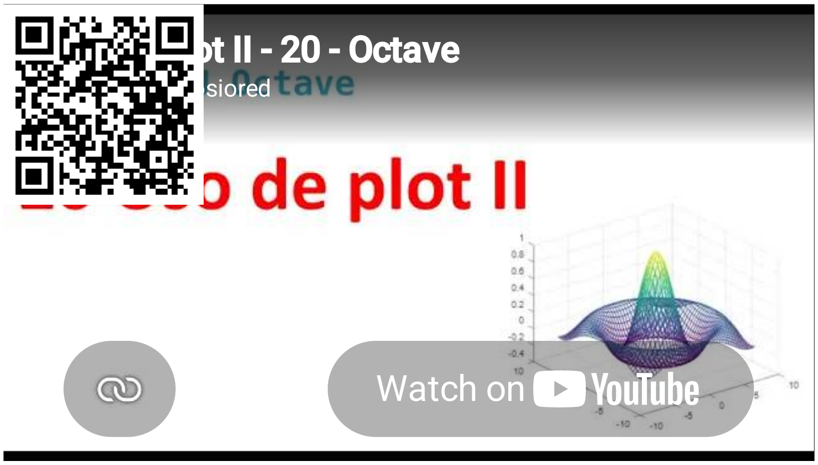
Ciclo for - 23 - Octave

clo for - 23 - Octave
siored
Ciclo for
Watch on YouTube

Plot - 19 - Octave


ot - 19 - Octave
siored
o de Plot
Watch on YouTube

Plot II - 20 - Octave


ot II - 20 - Octave
siored
tave
o de plot II
Watch on YouTube

This page titled 3.12: Chapter 3 for loop Videos in English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

3.13: Exercises

Before you go on, you might want to work on the following exercises.

? Exercise 3.5

Years ago I was in a fudge shop and saw a sign that said "Buy one pound of fudge, get another quarter pound free." That's simple enough.

But if I ran the fudge shop, I would offer a special deal to anyone who could solve the following problem:

> *If you buy a pound of fudge, we'll give you another quarter pound free. And then we'll give you a quarter of a quarter pound, or one-sixteenth. And then we'll give you a quarter of that, and so on. How much fudge would you get in total?*

Write a script called *fudge.m* that solves this problem. Hint: start with *series.m* and generalize it by replacing the ratio `1/2` with a variable, `r` .

? Exercise 3.6

We have already seen the Fibonacci sequence, F, which is defined recurrently as

$$\text{for } i \geq 3, \quad F_i = F_{i-1} + F_{i-2}$$

In order to get started, you have to specify the first two elements, but once you have those, you can compute the rest. The most common Fibonacci sequence starts with $F_1 = 1$ and $F_2 = 1$.

Write a script called *fibonacci2.m* that uses a `for` loop to compute the first 10 elements of this Fibonacci sequence. As a postcondition, your script should assign the 10th element to `ans` .

Now generalize your script so that it computes the nth element for any value of `n` , with the precondition that you have to set `n` before you run the script. To keep things simple for now, you can assume that `n` is greater than 0.

Hint: you'll have to use two variables to keep track of the previous two elements of the sequence. You might want to call them `prev1` and `prev2` . Initially, `prev1` $= F_1$ and `prev2` $= F_2$. At the end of the loop, you'll have to update `prev1` and `prev2` ; think carefully about the order of the updates!

This page titled 3.13: Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **3.11: Exercises** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

CHAPTER OVERVIEW

4: Vectors

4.1: Creating Vectors
4.2: Methods to Create Vectors in MATLAB
4.3: Vector Arithmetic
4.4: Selecting Elements
4.5: Indexing Errors
4.6: Vectors and Sequences
4.7: disp() and num2str() Functions and Concatenation
4.8: Plotting Vectors
  4.8.1: Plotting Vectors Additional Material
4.9: Recursive for Loops
4.10: Common Vector Operations
4.11: Common Vector Functions
4.12: Chapter Review
4.13: Chapter 4 Vector Videos In English and Espanol
4.14: Exercises

Most of the material in this chapter comes from Alan B. Downey, "Physical Modeling in Matlab", chapter 4

This page titled 4: Vectors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

4.1: Creating Vectors

A *vector* in MATLAB is a sequence of numbers. There are several ways to create vectors; one of the most common is to put a sequence of numbers in square brackets:

```
>> [1 2 3]
ans = 1     2     3
```

Another way to create a vector is the colon operator, which we have already used to create a range of values in a `for` loop.

```
>> 1:3
ans = 1     2     3
```

In general, anything you can do with a number, you can also do with a vector. For example, you can assign a vector value to a variable:

```
>> X = [1 2 3]
X = 1     2     3
```

Note that variables that contain vectors are often capital letters. That's just a convention; MATLAB doesn't require it, but it's a useful way to remember which variables are vectors.

This page titled 4.1: Creating Vectors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.1: Creating Vectors** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

4.2: Methods to Create Vectors in MATLAB

By Carey A. Smith

Example 4.2.1 Methods to create vectors in MATLAB

The following code is aloso in the attached file "Methods_to_create_vectors_in_Matlab.mlx".

```
%% Methods_to_create_vectors_in_Matlab.mlx
% Vectors can be created in Matlab in the following ways.

%% 1. Write a list of values between square brackets:
A = [ 0, 1, 4, 9, 25] % The commas are optional
B = [-3.1 1.789 2/7, exp(2)] % The values can be integers, decimals, or expressions.

%% 2. Use the colon operator to automatically create a vector of equally spaced values:
C = 1:10 % = [1 2 3 4 5 6 7 8 9 10]
% v = n1:n2 creates a vector from n1 to n2
% with an increment of 1
% An “increment” can define the spacing.
% The increment goes in the middle.
D = 0 : 2 : 10 % [0 2 4 6 8 10]
% The increment can be a fraction or a negative number:
E = 2 : 0.5 : 4 % [2 2.5 3 3.5 4]
F = 5 : -1 : 1 % [5 4 3 2 1]

% Alternate method: Multiply by a scale factor:
% Use the factor that multiplying a vector by a scalar
% multiplies each element of the vector by that factor.
theta = (0:(1/6):1) * pi
% theta = [0 0.5236 1.0472 1.5708 2.0944 2.618 3.1416]

%% 3. Derive a vector from another vector
v1 = 11:15 % [11 12 13 14 15]
% Ceate the squares of each element of v1
v2 = v1.^2 % [121 144 169 196 225]
% This uses the element-wise dot operator .^

%% 4. Concatenate vectors (merge 2 or more vectors into 1 vector)
name1 = 'Angela'
name2 = 'Merkel'
name12 = [name1,name2] % 'AngelaMerkel'
% You can add a space between the names:
name1_2 = [name1,' ',name2] % 'Angela Merkel'

%% 5. Use the zeros() function
G = zeros(1,4) % [0 0 0 0] 1 row, 4 columns
% The format is zeros(nrows, ncolumns)
% This is used to initialize a vector before a for loop

% 6. Use the ones() function
H = ones(1,4) % [1 1 1 1] 1 row, 4 columns
J = 3*ones(1,4) % [3 3 3 3] 1 row, 4 columns
% The format is ones(nrows, ncolumns)
% 7. Use the linspace command
K = linspace(1,10,5) % [1.00 3.25 5.50 7.75 10.00]
% This creates a vector of 5 evenly spaced values between 1 and 10
```

```
% 8. Use the logspace command
L = logspace(0,3,7) % 1.0E3*[0.001 0.0032 0.010 0.0316 0.100 0.3162 1.00]
% logspace(X1, X2, N) generates a row vector of N logarithmically
% equally spaced points between 10^X1 and 10^X2.
```

Solution

Add example text here.

Arreglos vectores - 5 - Octave

https://www.youtube.com/watch?v=H-5cicDVr5U&list=PLM-p96nOrGcYsb5oML9JunGDgduyt-0gb&index=5

This page titled 4.2: Methods to Create Vectors in MATLAB is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

4.3: Vector Arithmetic

4.3.1: Allen B. Downey's text:

As with numbers, you can do arithmetic with vectors. If you add a number to a vector, MATLAB increments each element of the vector:

```
X = 1:3
Y = X + 5
% Y = 6      7      8
```

The result is a new vector; the original value of `X` has not changed.

If you add two vectors, MATLAB adds the corresponding elements of each vector and creates a new vector that contains the sums:

```
Z = X + Y
% Z = 7      9     11
```

But adding vectors only works if the operands are the same size. Otherwise you get an error:

```
>> W = [1 2]
W = 1      2

>> X + W
Matrix dimensions must agree.
```

This error message might be confusing, because we think of `X` and `W` as vectors, not matrices. But in MATLAB, a vector is a kind of matrix. We'll come back to matrices later, but in the meantime, you might see the term in error messages and documentation.

If you divide two vectors, you might be surprised by the result:

```
>> X / Y
ans = 0.2953
```

MATLAB is performing an operation called right division, which is not what we expected. To divide the elements of `X` by the elements of `Y`, you have to use `./`, which is element-wise division:

```
>> X ./ Y
ans = 0.1667    0.2857    0.3750
```

Multiplication has the same problem. If you use `*`, MATLAB does matrix multiplication. With these two vectors, matrix multiplication is not defined, so you get an error:

```
>> X * Y
Error using  *
Incorrect dimensions for matrix multiplication.
Check that the number of columns in the first matrix
matches the number of rows in the second matrix.
To perform element-wise multiplication, use '.*'.
```

In this case, the error message is pretty helpful. As it suggests, you can use `.*` to perform element-wise multiplication:

```
>> X .* Y
ans = 6     14     24
```

As an exercise, see what happens if you use the exponentiation operator (`^`) with a vector.

4.3.2: Carey Smith's additional notes:

In both MATLAB and Octave, all of these vector multiplication expressions are valid and give the same results:

```
X.*Y   % No space between X and .* and no space between .* and Y
X.* Y  % No space between X and .* but a space between .* and Y
X .* Y % A space between X and .* and a space between .* and Y
```

But this expression fails:

```
X .*Y  % A space between X and .* but no space between .* and Y
```

In both MATLAB and Octave, all of these vector division expressions are valid and give the same results:

```
X./Y   % No space between X and ./ and no space between ./ and Y
X./ Y  % No space between X and ./ but a space between ./ and Y
X ./ Y % A space between X and ./ and a space between ./ and Y
```

But this expression fails:

```
X ./Y  % A space between X and ./ but no space between ./ and Y
```

In both MATLAB and Octave, all of these vector exponential expressions are valid and give the same results:

```
X.^3   % No space between X and .^ and no space between .^ and Y
X.^ 3  % No space between X and .^ but a space between .^ and Y
X .^ 3 % A space between X and .^ and a space between .^ and Y
```

But this expression fails:

```
X .^3  % A space between X and .^ but no space between .^ and Y
```

This page titled 4.3: Vector Arithmetic is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.2: Vector Arithmetic** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

4.4: Selecting Elements

You can select an element from a vector with parentheses:

```
>> Y = [6 7 8 9]
Y = 6     7      8      9

>> Y(1)
ans = 6

>> Y(4)
ans = 9
```

This means that the first element of `Y` is `6` and the fourth element is `9`. The number in parentheses is called the *index* because it indicates which element of the vector you want.

The index can be a variable name or a mathematical expression:

```
>> i = 1;
>> Y(i)
ans = 6
>> Y(i+1)
ans = 7
```

We can use a loop to display the elements of `Y`:

```
for i=1:4
     Y(i)
end
```

Each time through the loop we use a different value of `i` as an index into `Y`.

In the previous example we had to know the number of elements in `Y`. We can make it more general by using the `length` function, which returns the number of elements in a vector:

```
for i=1:length(Y)
     Y(i)
end
```

This version works for a vector of any length.

This page titled 4.4: Selecting Elements is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.3: Selecting Elements** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

4.5: Indexing Errors

An index can be any kind of expression, but the value of the expression has to be a positive integer, and it has to be less than or equal to the length of the vector. If it's zero or negative, you'll get an error:

```
>> Y(0)
Array indices must be positive integers or logical values.
```

If it's not an integer, you get an error:

```
>> Y(1.5)
Array indices must be positive integers or logical values.
```

If the index is too big, you also get an error:

```
>> Y(5)
Index exceeds the number of array elements (4).
```

The error messages use the word *array* rather than *matrix*, but they mean the same thing, at least for now.

This page titled 4.5: Indexing Errors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.4: Indexing Errors** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

4.6: Vectors and Sequences

Vectors and sequences go together nicely. For example, another way to evaluate the Fibonacci sequence from Chapter 2 is to store successive values in a vector. Remember that the definition of the Fibonacci sequence is $F_1 = 1$, $F_2 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i > 2$.

Listing 4.1 shows how we can compute the elements of this sequence and store them in a vector, using a capital letter for the vector `F` and lower-case letters for the integers `i` and `n`.

Listing 4.1: Calculating the Fibonacci sequence using a vector

```
F(1) = 1
F(2) = 1
for i=3:n
    F(i) = F(i-1) + F(i-2)
end
```

If you had any trouble with , you have to appreciate the simplicity of this script. The MATLAB syntax is similar to the math notation, which makes it easier to check for correctness.

If you only want the nth Fibonacci number, storing the whole sequence wastes some space. But if wasting space makes your code easier to write and debug, that's probably okay.

This page titled 4.6: Vectors and Sequences is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.5: Vectors and Sequences** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

4.7: disp() and num2str() Functions and Concatenation

By Carey Smith.

4.7.1: disp()

The disp() function can display a number variable or a character string.

✓ Example 4.7.1 disp() a character string or a number

```
% Display a character string:
disp('theta')
```

The output is:

```
theta
```

```
% Display a number variable
theta = 2*pi;
disp(theta)
```

The output is:

```
6.2832
```

Solution

Add example text here.

To display a message that has a quote mark, you need to enter 2 quotes marks, like this:
disp('Lincoln"s birthday is Feb. 12')

4.7.2: Concatenation

Definition: Concatenation

Concatenation: Connect 2 or more vectors or character strings into a single vector or character string.

✓ Example 4.7.2 Concatenation of 2 vectors

```
a = 1:4
b = [12, 13, 14]
% Put 2 or more items of the same type within square brackets to concatenate them:
ab = [a, b]
```

Result:

```
[1 2 3 4 12 13 14]
```

Solution

Add example text here.

.

✓ Example 4.7.3 **Concatenation of 2 character strings**

Concatenation of 2 character strings:

```
name1 = 'Frederick'
name2 = 'Douglas'
fullname = [name1,' ',name2]
```

Result:

```
'Frederick Douglas'
```

% The ' ' puts a space between the 2 names

Solution

Add example text here.

4.7.3: num2str()

In order to concatenate a character string with a number, the number needs to be converted to a character string. That can be done with the num2str() function. num2str(x) converts the number variable, x, to a character string.

✓ Example 4.7.4

Examples:

```
x = 1/7

num2str(x) % This displays x with the default number of significant figures
ans =
    '0.14286'
num2str(x, 3) % This displays x with 3 significant figures

ans =
    '0.14286'
```

Solution

Add example text here.

Each digit in the number is converted tp a character using the ASCII standard. For details of the the ASCII standard, see https://en.Wikipedia.org/wiki/ASCII.

✓ Example 4.7.5 Concatenation of a character string with a number

Display the variable name 'theta' and its value (a number)

```
theta = 2*pi;
```

The 2 items need to be of the same type (format).

Theta's type is a double-precision number.

Character strings (text) are stored in ASCII format.

You do not need to know any details about ASCII format, except that it is different than number format. If you want to learn more, see:

https://en.wikipedia.org/wiki/ASCII

We convert the number to a character string using the num2str() function as follows:

```
theta_char_str = num2str(theta)
```

```
Result: '6.2832'
```

The next line of code concatenates the variable name and its value into a single character string:

```
theta_msg = ['theta = ', theta_char_str]
```

Result: 'theta = 6.2832'

We can combine displaying the name with the conversion of the number to a string in 1 line of code as shown here:

```
disp(['theta = ', num2str(theta)])
```

Same result: 'theta = 6.2832'

Solution

Add example text here.

.

? Exercise 4.7.1 Help on num2str

Part a: For this exercise you will use num2str() to disply pi with 3 significant figures and with with 10 significant figures. To do this, you need to read the help on the num2str() function. This also gives you practice using the help.

Method 1: In the command window, type:

```
help num2str
```

This works in either MATLAB or Octave.

Method 2 (MATLAB): enter "num2str" in either the editor or the command window, then right-click on num2str and select "Help on Selection". A window pops up that lets you open MATLAB's help browser with documentation and examples.

Method 2 (Octave): Under the command window click "Documentation". At the top, select the "Function Index" tab and type num2str in the search box.

Part b: Create an m-file. In this file, use num2str to display **pi** with 3 significant figures.

Part c: Also use num2str to display **pi** with 10 significant figures.

Save your file and submit it.

Answer

Add texts here. Do not delete this text first.

.

This page titled 4.7: disp() and num2str() Functions and Concatenation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

4.8: Plotting Vectors

If you call `plot` with a vector as an argument, MATLAB plots the indices on the x-axis and the elements on the y-axis. To plot the Fibonacci numbers we computed in Listing 4.1, we'd use

```
plot(F)
```

Figure 4.1 shows the result.

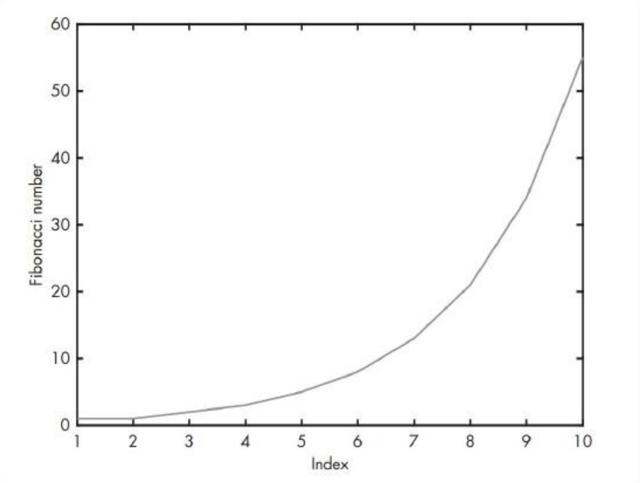


Figure 4.1: The first 10 elements of the Fibonacci sequence.

This way of looking at a vector is often useful for debugging, especially if it is big enough that displaying the elements on the screen is unwieldy.

This page titled 4.8: Plotting Vectors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.6: Plotting Vectors** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

4.8.1: Plotting Vectors Additional Material

Watch this video on making an x-y plot with vectors. It also shows how to plot 2 curves on 1 figure with "hold on".

MATLAB Function Plotting Basics, Robert Talbert

ATLAB Function Plotting Basics
ertTalbertPhD
Command Window
Workspace
Command History
Watch on YouTube

Important note:

When a function has more than 1 input, you must put commas between the inputs.

Even though you can leave-out commas when specifying a vector such as x = [1 2 5 8], this does not work for functions. The commas between the inputs are required.

Examples:

plot(x, y), not plot(x y)

plot(x, y, '*'), not plot(x y '*')

.

Additional MATLAB functions for labeling plots:

title() puts a title on a plot. The argument of title is a character string.

Examples:

title('sin(x) vs. t')

xlabel() and ylabel() put labels on the x and y axes of a plot.

Examples:

xlabel('Time (s)')

ylabel('Distance')

? Exercise 4.8.1.1 Voltage_Vector

(1 pt) Create a Matlab script: voltage_vector_YourName.m
(1 pt) Clear any variables; close any figures; eliminate white space; clear the console

```
clear all; close all; format compact; clc;
```

%% V = IR

(2 pts) Create a vector I (current) from 0 to 15 amps in steps of 3 amps

(2 pts) Create a vector R (resistance) from 100 to 600 Ohms in steps of 100 Ohms

(2 pts) Compute the corresponding vector of voltages by multiplying I time R element-wise with this code

```
V = I.*R % (volts)
```

%% Plot

% (1 pt) Open a figure window

% (1 pt) Plot V (y-axis) vs. I (x-axis) with this code:

```
plot(I, V, '-o')
```

```
title('Voltage vs. Current')
```

% (1 pt) Label the x-axis with this code:

```
xlabel('I (amps)')
```

% (1 pt) Label the y-axis with this code:

```
ylabel('V (volts)')
```

Answer

The solution is not given here.

.

This page titled 4.8.1: Plotting Vectors Additional Material is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

4.9: Recursive for Loops

By Carey A. Smith

A "recursive variable" in MATLAB is one whose updated value is a function of it's previous value.

A "recursive for loop" is one in which the calculations for each iteration depend on the results of the previous iteration. It can have one or more recursive variables.

The Fibonacci sequence in section 4.6 is an example of a recursive for loop, because each term is the sum of the 2 previous values in the sequence.

A power sequence is another example of a recursive for loop.

✓ Example $4.9.1$ Power Sequence

```
% Initialize these variables:
n = 6;    % The number of terms in the sequence
A0 = 4; % The first value
r = 1/2;  % The ratio of successive terms
% Write a for loop that computes the terms in the sequence
% which are computed with this expression:
%   A(k) = A0*r^k;
for k = 1:n
  A(k) = A0*r^k;
end
A % This diaplays all the values of the vector A
```

Solution

Add example text here.

.

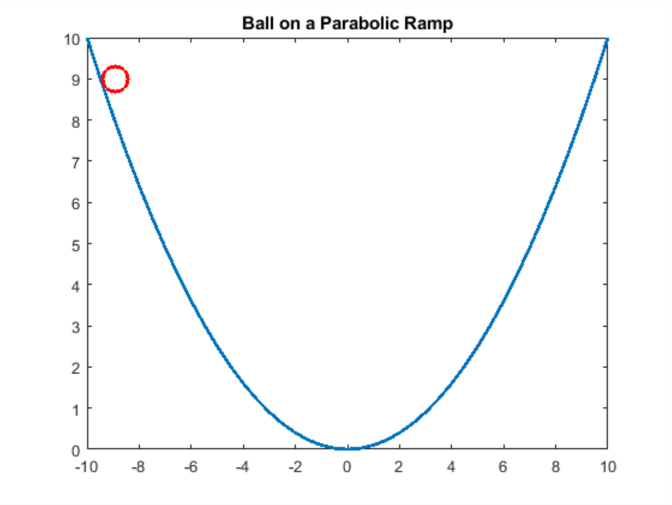
This is one example of a recursive loop. The height of the ball is the recursive variable.

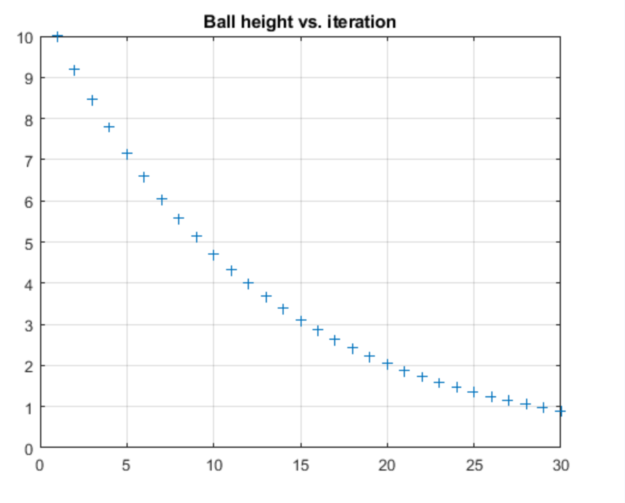
✓ Example $4.9.1$ Ball Rolling on a Parabola with Friction

```
% A ball starts at height h1 on the ramp:
h1 = 10 % cm Initial height on the ramp
% Each time it rolls back and forth, it loses some energy due to friction
ff = 0.08 % The fraction of friction lost each iteration
% Use a for loop to compute the height on successive iterations.
% Let the number of iterations be:
n = 30
% Store the heights of each iteration in a vector.
% Initialize this vector to all zeros
h = zeros(1,n) % a row vector that is 8 long.
h(1) = h1
for k = 1:(n-1)
    h_loss = ff*h(k);
    h(k+1) = h(k) - h_loss;
end
%% Draw a parabolic ramp
x = -10:0.5:10;
y = (0.1)*x.^2;
figure;
plot(x,y,'LineWidth',2)

% Add a ball
bxc = -8.9; % ball center
byc = 9; % ball center
hold on;
plot(bxc, byc,'ro','MarkerSize',16,'LineWidth',2)
title('Ball on a Parabolic Ramp')
%% Plot the vector of heights with '+' markers
figure
plot(h, '+')
grid on
title('height vs, iteration')
ylim([0,h1])
```

Solution

Add example text here.





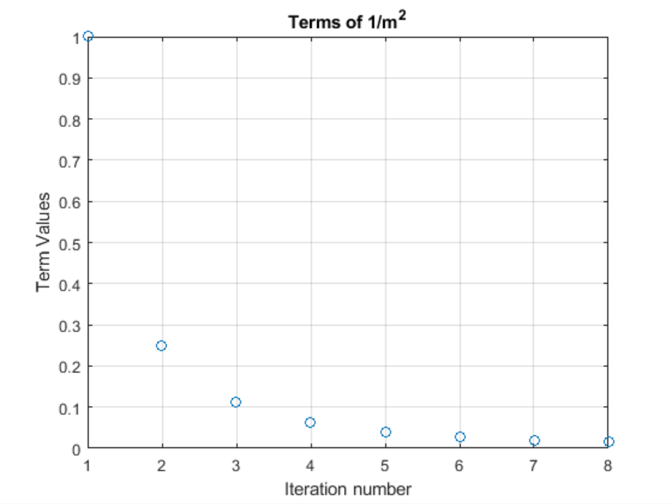
.

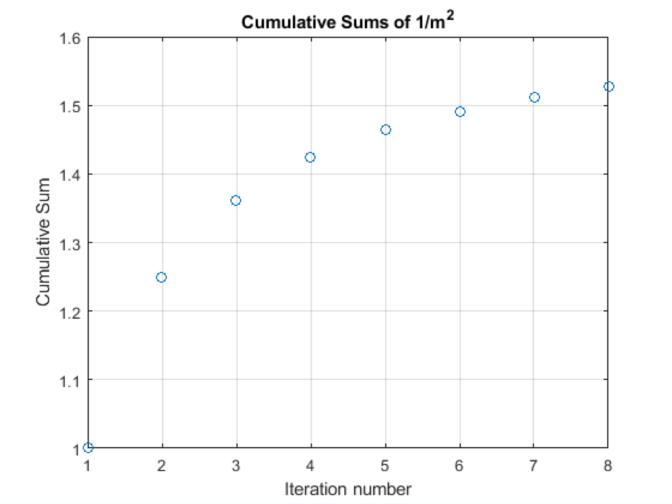
The summation of a series is an example of a recursive loop. The total is a recursive variable.

Example 4.9.2 Cumulative Sum Vector Plot

```
format compact; format short g;
clear all; close all; clc;
%% Initialize the term and total vectors
term = zeros(1,8);    % Initialize the term vector
total = zeros(1,8);   % Initialize the total vector
term(1) = 1/1^2;
total(1)= term(1);
for m = 2:8
  term(m) = 1/m^2;
  total(m) = total(m-1) + term(m);
end
%% Display the final value
disp(['The final sum= ',num2str(total(end))])
% Display the last term
disp(['The last term= ',num2str(term(end))])

%% Plot the terms
figure;
plot(term,'o')
grid on;
title('Terms of 1/m^2')
xlabel('Iteration number')
ylabel('Term Values')

%% Plot the cumulative sums
figure;
plot(total,'o')
grid on;
title('Cumulative Sums of 1/m^2')
xlabel('Iteration number')
ylabel('Cumulative Sum')
```

Answer





.

.

The following recursive sum example compares the code for a scalar total (the total is a single value) to the code for a vector total (the total vector includes the totals for each iteration).

When there are 2 data sets plotted on 1 figure, you can use the legend() function that identifies each data set, as shown in the following example.

It also shows the use of the text() function to place information on the figure. This is used in combination with the num2str() function to convert a computed value to a character string that can become part of a title or text, using concatenation of character strings.

✓ Example $4.9.3$ Scalar vs. Vector Sums

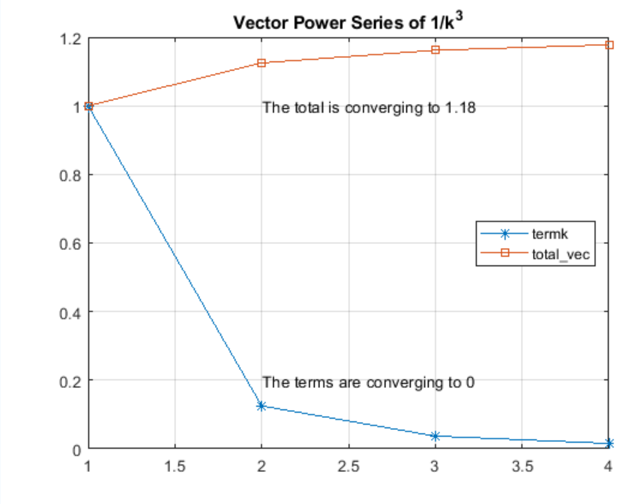
```
%% Scalar loop sum
n = 4; % n = the number of terms in the series
total_scalar = 0; % This total is updated on each iteration
for k = 1:n
    term = 1/k^3;
```

```
   % Sum the series using either of these 2 methods:
   % cum_sum(m) = cum_sum(m-1) + term(m);  % Method 1

   cum_sum_new = cum_sum(m-1) + term(m);  % Method 2

   cum_sum(m) = cum_sum_new;

    total_scalar = total_scalar + term;   % Method 1
    % Update total by adding the current term to the previous value of total.
    % Recall that "=" means assignment, not equality.

    total_new = total_scalar + term;   % Method 2
    total_scalar = total_new;
end
total_scalar % echo the final total
% total_scalar = 1.1777
% This is a scalar loop sum, because termk and total are single values which change
on each iteration

%% Vector loop sum
n = 4; % n = the number of terms in the series
total_vec = zeros(1,n); % This total is updated on each iteration
% total_vec(k) will be the partial sum of the terms up thru the kth term.
% Initial the 1st term and the total:
k = 1;
termk(k) = 1/k^3;
total_vec(k) = termk(k);
for k = 2:n % Start with k=2, becuz the loop uses total_vec(k-1)
    termk(k) = 1/k^3;
    total_vec(k) = total_vec(k-1) + termk(k);
    % Update total by adding the current term to the previous value of total.
    % New value is total(k). Previous value is total(k-1)
    % Recall that "=" means assignment, not equality.
end
total_vec % Echo total_vec

%% Since termk and total_vec are vectors, we can plot them after the for loop.
kk = 1:n; % This is the x-axis vector
figure;
plot(kk, termk, '-*')
hold on;
plot(kk, total_vec, '-s')
grid on;
legend('termk','total\_vec','Location','East')
title('Vector Power Series of 1/k^3')
 If you are using MATLAB, add this code:
   total_end = round(total_vector(end),2) % Round to 2 decimal places

 If you are using Octave, add this code:

   total_end = total_vector(end)  % Don't round

text(2,1,['The total is converging to ',total_str]) % concatenate the words with the
number
text(2,0.2,'The terms are converging to 0')
```

Solution

% total_scalar = 1.1777

% total_vec = 1 1.125 1.162 1.1777



.

.

Population growth is another example of a recursive for loop.

✓ Example 4.9.4 Population Growth of Chipmunks

```
Figure \(\PageIndex{i}\): https://commons.wikimedia.org/wiki/File:Chipmunk_in_PP_
Creative Commons Attribution-Share Alike 4.0 International license.
```

% Population growth of chipmunks

```
n = 5 % The number of years (iterations)
```

% Create this vector:

```
chipmunks = zeros(1,n);
```

% The population starts with 40 individuals, so set the 1st value = 40.

```
chipmunks(1) = 40;
```

% Assume that there are approximately equal numbers of males and females.

% Start a for loop that models the size of this population for 5 years.
% Let the for loop index be k; k will go from 1 to (n-1),
% because it computes the population for year k+1, based on year k.

```
for k = 1:(n-1)
    % Inside the for loop do the following to update the number of chipmunks
    %  for each for successive year:
    % Each year, on average, 40% of the chipmunks die, so compute:
     died = round(0.40*chipmunks(k)) % chipmunks(k) is the number at the beginning
of year k.
    % The round() function makes sure that the result is an integer.
    % Subtract the number that died from the population.
    %  Let the resulting number be called survivors.
    survivors = chipmunks(k) - died
    % Compute an average of 3 baby chipmunks for each pair of surviving chipmunks
    %  and round this value:
    babies = round(3*(survivors/2))
    % Add the babies to the surviving foxes
    %  to get the new total population for the start of the next year:
  chipmunks(k+1) = survivors + babies
end % end the for loop
```

% After the loop end, open a new figure and plot the number of chipmunks each year.
% Use the ‘v’ marker for each data point.

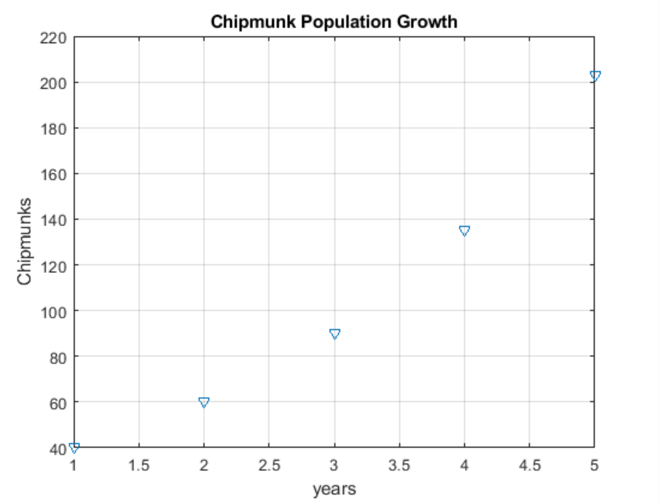
```
figure
plot(chipmunks,'v')
```

% (1 pt) Turn the grid on and add a title and axis labels:

```
grid on
title('Chipmunk Population Growth')
xlabel('years')
ylabel('Chipmunks')
```

Solution

Add example text here.



Many variations on the population growth example are possible.

? Exercise 4.9.1 Modified Population Growth

Modify the chipmunk population growth example by changing the death rate and/or the birth rate, such that the number of chipmunks after 5 years is a value between 1.5 and 3 times the initial population.

This will require some trial and error.

Answer

Figure this out on your own.

.

There is a potential problem that you must avoid. If you create a program that mistakenly calls itself, this creates an unintentional recursive infinite loop. It may only stop when the maximum number of recursive calls allowed by MATLAB/Octave is reached (about 500 calls) or if you kill the program either by closing MATLAB/Octave or keying Ctrl-C. When a student writes a script that calls itself, it is almost always a mistake. However, there are a very few valid, specialized functions that use recursive calls to themselves, such as code search for free memory to be allocated. Such code must have logic to end the self-calling recursion.

.

This page titled 4.9: Recursive for Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

4.10: Common Vector Operations

We've covered some of the basic features of vectors. Let's now look at some common patterns we use to work with data stored in vectors.

Reduce

We frequently use loops to run through the elements of a vector and add them up, multiply them together, compute the sum of their squares, and so on. This kind of operation is called *reduce*, because it reduces a vector with multiple elements down to a single number.

For example, the loop in Listing 4.2 adds up the elements of a vector named `X` (which we assume has been defined).

List 4.2: Reducing a vector to a single scalar value (the sum)

```
total = 0
for i=1:length(X)
    total = total + X(i)
end
ans = total
```

The use of `total` as an accumulator is similar to what we saw in Chapter 3. Again, we use the `length` function to find the upper bound of the range, so this loop will work regardless of the length of `X`. Each time through the loop, we add in the `i` th element of `X`, so at the end of the loop `total` contains the sum of the elements.

MATLAB provides functions that perform some reduce operations. For example, the `sum` function computes the sum of the elements in a vector, and `prod` computes the product.

Apply

Another common use of a loop is to run through the elements of a vector, perform some operation on the elements, and create a new vector with the results. This operation is called *apply*, because you apply the operation to each element in the vector.

For example, the loop in Listing 4.3 creates a vector `Y` that contains the squares of the elements of `X` (assuming, again, that `X` is already defined).

Listing 4.3: Making a new vector Y by squaring the elements in X

```
for i=1:length(X)
    Y(i) = X(i)^2
end
```

Many apply operations can be done with element-wise operators. The following statement is more concise than the loop in Listing 4.3.

```
Y = X .^ 2
```

It also runs faster!

This page titled 4.10: Common Vector Operations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.7: Common Vector Operations** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

4.11: Common Vector Functions

By Carey A. Smith

Additional MATLAB vector functions:

✓ Example 4.11.1 Functions that compute a single number from a vector

```
% This is vector of CO2 levels (ppm) measured at Mauna Loa Hawaii in 2018:
CO2  = [407.96,  408.32,  409.39,  410.25,  411.24,  410.79,  408.70,  406.97,  405.52,
406.00, 408.02, 409.08];
% Reduce functions:
CO2_sum = sum(CO2) % Total
CO2_length = length(CO2) % The number of items in the vector
CO2_min = min(CO2) % Minimum
CO2_max = max(CO2) % Maximum
```

```
Note: numel(x) give the same answer as length(x) when x is a vector. For a a matrix
or array, they give different answers. I this book, length will only be used for
vectors.
```

Solution

N/A

✓ Example 4.11.1 Functions that apply an operation to a vector

This is vector of CO2 levels (ppm) measured at Mauna Loa Hawaii in 2018:

```
CO2  = [407.96,  408.32,  409.39,  410.25,  411.24,  410.79,  408.70,  406.97,  405.52,
406.00, 408.02, 409.08];
% Apply functions and operators:
CO2_sqrt = sqrt(CO2) % Square roots of the elements
CO2_square = CO2.^2 % Squares of the elements (element-wise)
CO2_log10 = log10(CO2) % logarithm, base 10
CO2_log = log(CO2) % logarithm, base e (natural logarithm)
% log() should be ln(), but this is a long tradition in programming languages.
CO2_exp = exp(CO2) % e^x
```

```
CO2_column=CO2' % The ' symbol transposes a  vector
```

Note that sin(theta), cos(theta), tan(theta) are for theta in radians.

```
theta = pi*(0 : 0.1 : 0.3)
theta_sin = sin(theta)
theta_cos = cos(theta)
theta_tan = tan(theta)
```

Note that sind(alpha), cosd(alpha), tand(alpha) are for alpha in degrees.

```
alpha = 0 : 15 : 45
sin_alpha = sind(alpha)
cos_alpha = cosd(alpha)
tan_alpha = tand(alpha)
```

The attached file, MatlabCommands_Brian_Vick.pdf, courtesy of Dr. Brian Vick from Virginia Tech, is a very useful summary of the the most common MATLAB commands.

Solution

Add example text here.

? Exercise 4.11.1 Squareroot of the sum of squares

Create this vector: v = [3, 4, 5, 6, 7]

Then write 1 line of cose that computes the square root of the sum of the squares of the elements of the vector, without using a loop.Write 1 line of code that

Answer

Use the vector operator v.^2 to square each element of the vector; sum the elements sum(); compute the squareroot.

? Exercise 4.11.1 sin() of a vector

(1 pt) Create a Matlab script: sin_vector_YourName.m
(1 pt) Clear any variables; close any figures; eliminate white space; clear the console with this line of code:
clear all; close all; format compact; clc;

%% Part 1
(1 pt) Create a vector called degrees = 0 to 90 in steps of 10 degrees
degrees = 0 : 10 : 90;

(2 pts) Use sind() to compute the sines of the degrees vector.

(2 pts) Create a 2nd vector: radians = (2*pi/360)*degrees.
(2 pts) Use sin() to compute the sines of the radians vector.
%% Part 2
(2 pts) Convert the radians vector created in part 1 from a row vector
to a column vector with this code:
radians_col = radians' % The ' transposes a vector
(1 pt) Use cos() to compute the cosine this column radians vector.

Answer

Add texts here. Do not delete this text first.

.

Link to additional function information

In the "Back Matter" of this book, there is a section called "MATLAB Commands and Functions". This has an attachment with short description of many MATLAB command and functions. This is a good place to search for a function that would be useful for a particular program.

MATLAB Functions, Commands and Special Characters

You can also use the MATLAB help browser to find useful functions.

.

This page titled 4.11: Common Vector Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

4.12: Chapter Review

In this chapter, we used a vector to store the elements of a sequence. We learned how to select elements from a vector and perform vector arithmetic. We performed reduce and apply operations using `for` loops, MATLAB functions, and element-wise operations.

Here are some terms from this chapter you might want to remember.

A *vector* is a sequence of values, which is a kind of *matrix*, also called an *array* in some MATLAB documentation.

An *index* is an integer value used to indicate one of the elements in a vector or matrix (also called a *subscript* in some MATLAB documentation).

An operation is *element-wise* if it acts on the individual elements of a vector or matrix (unlike some linear algebra operations).

You can *apply* an operation to all elements of a vector, and you can a vector to a single value, for example by computing the sum of the elements.

In the next chapter, we'll meet the most important idea in computer programming: functions!

This page titled 4.12: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.8: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

4.13: Chapter 4 Vector Videos In English and Espanol

English

Matlab plot basics + hold video, RTalbert

ATLAB Function Plotting Basics
ertTalbertPhD
Command Window
Watch on YouTube

Espanol

Arreglos vectores - 5 - Octave

reglos vectores - 5 - Octave
siored
reglos
Vectores
Watch on YouTube

Recombinar vectores - 8 - Octave

combinar vectores - 8 - Octave
siored
ecombinar
vectores
Watch on YouTube

This page titled 4.13: Chapter 4 Vector Videos In English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

4.14: Exercises

Before you go on, you might want to work on the following exercises.

? Exercise 4.1

Write a loop that computes the first n elements of the geometric sequence $A_{i+1} = A_i/2$ with $A_1 = 1$. Notice that math notation puts A_{i+1} on the left side of the equality. When you translate to MATLAB, you might want to rewrite it with A_i on the left side.

? Exercise 4.2

Write an expression that computes the square root of the sum of the squares of the elements of a vector, without using a loop.

? Exercise 4.3

The ratio of consecutive Fibonacci numbers, F_{n+1}/F_n, converges to a constant value as n increases. Write a script that computes a vector with the first n elements of a Fibonacci sequence (assuming that the variable `n` is defined) and then computes a new vector that contains the ratios of consecutive Fibonacci numbers. Plot this vector to see if it seems to converge. What value does it converge on?

? Exercise 4.4

The following set of equations is based on a famous example of a chaotic system, the Lorenz attractor (see https://greenteapress.com/matlab/lorenz):

$$\begin{aligned} x_{i+1} &= x_i + \sigma\left(y_i - x_i\right) dt \\ y_{i+1} &= y_i + \left[x_i(r - z_i) - y_i\right] dt \\ z_{i+1} &= z_i + \left(x_i y_i - b z_i\right) dt \end{aligned} \tag{4.14.1}$$

1. Write a script that computes the first 10 elements of the sequences X, Y, and Z and stores them in vectors named `X`, `Y`, and `Z`.

   Use the initial values $X_1 = 1$, $Y_1 = 2$, and $Z_1 = 3$ with the values $\sigma = 10$, $b = 8/3$, $r = 28$, and $dt = 0.01$.
2. Read the documentation for `plot3` and `comet3`, and plot the results in three dimensions.
3. Once the code is working, use semicolons to suppress the output and then run the program with sequence lengths of 100, 1,000, and 10,000.
4. Run the program again with different starting conditions. What effect does it have on the result?
5. Run the program with different values for σ, b, and r, and see if you can get a sense of how each variable affects the system.

? Exercise 4.5

The logistic map (see https://greenteapress.com/matlab/logistic) is described by the following equation:

$$X_{i+1} = rX_i(1 - X_i)$$

where X_i is a number between 0 and 1, and r is a positive number.

1. Write a script named *logmap.m* that computes the first 50 elements of X with `r` = 3.9 and `X1` = 0.5, where `r` is the parameter of the logistic map and `X1` is the initial value.
2. Plot the results for a range of values of r from 2.4 to 4.0. How does the behavior of the system change as you vary r?

This page titled 4.14: Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith via source content that was edited to the style and standards of the LibreTexts platform.

- **4.9: Exercises** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

CHAPTER OVERVIEW

5: User-Defined Functions

5.1: Name Collisions
5.2: Defining Functions
  5.2.1: Function Case Study (Lambert)
5.3: Function Documentation
5.4: Naming Functions
  5.4.1: Function Names and MATLAB Keywords
5.5: Functions With Multiple Input Variables
  5.5.1: Multiple Inputs Examples and Exercises
  5.5.2: Mutiple Inputs Example- Ideal Gas law (Lambert)
5.6: Functions with Multiple Outputs
5.7: Matlab Functions with No Inputs
5.8: Matlab Functions with No Outputs
5.9: Matlab Functions with No Inputs and No Outputs
5.10: Functions with variable numbers of inputs or outputs
5.11: How Functions Work
5.12: Chapter Review
5.13: Chapter 5 Function videos English and Espanol
5.14: Exercise

Most of the material in this chapter comes from Downey, chapter 5 and Lambert, chapter 8.

This page titled 5: User-Defined Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.1: Name Collisions

All scripts run in the same workspace, so if one script changes the value of a variable, all other scripts see the change. With a small number of simple scripts, that's not a problem, but eventually the interactions between scripts become unmanageable.

For example, the following script computes the sum of the first `n` terms in a geometric sequence, but it also has the *side effect* of assigning values to `A1`, `total`, `i`, and `a`.

```
A1 = 1;
total = 0;
for i=1:10
    a = A1 * 0.5^(i-1);
    total = total + a;
end
ans = total
```

If you were using any of those variable names before calling this script, you might be surprised to find, after running the script, that their values had changed. If you have two scripts that use the same variable names, you might find that they work separately and then break when you try to combine them. This kind of interaction is called a *name collision*.

As the number of scripts you write increases, and they get longer and more complex, name collisions become more of a problem. Avoiding this problem is one of several motivations for functions.

This page titled 5.1: Name Collisions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **5.1: Name Collisions** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

5.2: Defining Functions

A *function* is like a script, except that each function has its own workspace, so any variables defined inside a function only exist while the function is running and don’t interfere with variables in other workspaces, even if they have the same name. Function inputs and outputs are defined carefully to avoid unexpected interactions.

To define a new function, you create an M-file with the name you want and put a function definition in it. For example, to create a function named `myfunc`, create an M-file named *myfunc.m* and put the following definition into it (Listing 5.1):

List 5.1: A function definition

```
function res = myfunc(x)
    s = sin(x)
    c = cos(x)
    res = abs(s) + abs(c)
end
```

The first non-comment word of the file has to be `function`, because that’s how MATLAB tells the difference between a script and a function file.

A function definition is a compound statement. The first line is called the *signature* of the function; it defines the inputs and outputs of the function. In Listing 5.1 the *input variable* is named `x`. When this function is called, the argument provided by the user will be assigned to `x`.

The *output variable* is named `res`, which is short for *result*. You can call the output variable whatever you want, but as a convention, I like to call it `res`. Usually the last thing a function does is assign a value to the output .

Once you’ve defined a new function, you call it the same way you call built-in MATLAB functions. If you call the function as a statement, MATLAB puts the result into `ans`:

```
>> myfunc(1)

s = 0.84147098480790

c = 0.54030230586814

res = 1.38177329067604

ans = 1.38177329067604
```

But it’s more common (and better style) to assign the result to a variable:

```
>> y = myfunc(1)

s = 0.84147098480790

c = 0.54030230586814

res = 1.38177329067604

y = 1.38177329067604
```

While you're debugging a new function, you might want to display intermediate results like this, but once it's working, you'll want to add semicolons to make it a *silent function*. A silent function computes a result but doesn't display anything (except sometimes warning messages). Most built-in functions are silent.

Each function has its own workspace, which is created when the function starts and destroyed when the function ends. If you try to access (read or write) the variables defined inside a function, you will find that they don't exist.

```
>> clear
>> y = myfunc(1);
>> who
Your variables are: y

>> s
Undefined function or variable 's'.
```

The only value from the function that you can access is the result, which in this case is assigned to `y`.

If you have variables named `s` or `c` in your workspace before you call `myfunc`, they will still be there when the function completes.

```
>> s = 1;
>> c = 1;
>> y = myfunc(1);
>> s, c

s = 1
c = 1
```

So inside a function you can use whatever variable names you want without worrying about collisions.

This page titled 5.2: Defining Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **5.2: Defining Functions** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

5.2.1: Function Case Study (Lambert)

By Adam Lambert, Zero to MATLAB, 8.1 Basic Functions

Generally, when we make a function we will be using two separate m-files. One m-file will contain the code for the function. A 2nd file (the test file) will call the function.

Two notes of caution:

1. Do not try to run a function file by itself. A function file is different from a script file. If you do try to run a function file, MATLAB or Octave will usually complain that the inputs are not defined. You need to call the function from the test script file.

2. Do not "clear all" inside the function. That would clear your inputs and the function will fail. Only "clear all" in the test script file.

The function syntax can be confusing at first. In simple problems it can seem pointless to implement a function, but you want to practice creating functions with simple problems. As you begin to write more complex programs, functions become a valuable tool. Take the time to learn the operation with these simple examples.

The syntax required to build a function is the most complex that we have encountered. It takes some practice to get it organized in your head. Let's dive into a simple example and take some time to discuss all of the pieces.

✓ Example 5.2.1.1 times_two function

This is a function which will accept a number as input, multiply it by two, and return the result of that calculation as output. This code should be saved in an m-file called times_two.m. The name of the function must be the same as the m-file where it is stored.

```
function y = times_two(x)
    y = 2*x;
end
```

On the first line we see the function command, followed by the expression y = times_two(x). We call this line the declaration. (Some authors call this the function declaration; others call it the function signature. Both declaration and signature mean the same thing.) The function command tells MATLAB that this code is a function. It must be the first command in the file.

The variable x is the input variable for the function. Whatever number that we pass to the function will be assigned to the variable x which can then be used in the calculations inside the function. The variable y to the left of the assignment operator is the output variable. This value must be assigned during the execution of the function. In this simple case, the assignment is the only expression inside the function. As long as the variable is assigned within the function, then the value of the variable will be returned once the code inside the function completes.

The name of the function, times_two, is also the command that we use to execute the function. When we execute a function, we say that we call the function. A line of code in a script that calls a function is often referred to as a function call. In order to call a function, the m-file must be saved in the current directory.

If you look at the structure of the declaration, you can see that it mimics the structure of the code which is used to call it. The output variable is in the position to be assigned a value, and the input variable is in the position to pass a value into the function. Consider this similar expression using a built-in command:

```
>>z = sin(pi)
```

The value of pi is passed into the sin function and the result is assigned to the variable z. The declaration of this function would look like this.

```
function y = sin(x)
```

Inside there would be an algorithm to calculate the sine of the value of x and assign it to the variable y.

Now that we've had an overview of how functions are made and implemented, let's practice with this simple example to learn more about the behavior.

1. Open up a new m-file and save it in the current directory as times two.m.

2. Type the code from the times two example into the script and save it. Note that the file name and the function name are the same.

3. Now go to the Command Window and call the function at the prompt. We need to input a number, just like if we used one of the trigonometric functions.

```
>>times two(4)
```

The function will return the value 8. Note that the variable y is not saved in the Workspace and the value is assigned to the default variable name ans. If you do have a variable y from a previous calculation then clear the Workspace and try again. This is important, because the function only returns a value, not a variable. We'll discuss this in more detail in the next section.

4. We can use a variable to pass the input value into the function. We can also use a variable to store the returned results. Try the following sequence in the Command Window.

```
>>a = 4;
>>b = times two(a);
```

Note that the variable names are not the same as the names inside the function. They do not have to be, because they are simply passing values in and out of the function. The external variables never interact with the code inside the function.

Let's look back at the code in times two.m and walk through the whole process. First, the input value is assigned to the variable x. This can occur with a number or a variable. If the input to the function is a variable, then MATLAB will look up the value and assign that value to the variable x. Once the variable x is defined, then the code will execute. In this case, x will be multiplied by 2 and the result assigned the to variable y. Then, because all of the code inside the function has completed, the value of the output variable will be returned.

This is the most complex topic that we have covered, so don't be discouraged if this is confusing the first time through. As we move through the chapter and you get more experience, the flow of information will be easier to see.

Solution

Add example text here.

This page titled 5.2.1: Function Case Study (Lambert) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.3: Function Documentation

At the beginning of every function file, you should include a comment that explains what the function does:

```
% res = myfunc(x)
% Compute the Manhattan distance from the origin to the
% point on the unit circle with angle (x) in radians.

function res = myfunc(x)
% this is not part of documentation given by help function

    s = sin(x);
    c = cos(x);
    res = abs(s) + abs(c);
end
```

When you ask for `help`, MATLAB prints the comment you provide.

```
>> help myfunc
  res = myfunc(x)
  Compute the Manhattan distance from the origin to the
  point on the unit circle with angle (x) in radians.
```

There are lots of conventions about what should be included in these comments. Among other things, it’s a good idea to include the following:

Signature

The signature of the function, which includes the name of the function, the input variable(s), and the output variable(s).

Description

A clear, concise, abstract description of what the function does. An *abstract* description is one that leaves out the details of *how* the function works and includes only information that someone using the function needs to know. You can put additional comments inside the function that explain the details.

Variables

An explanation of what the input variables mean; for example, in this case it is important to note that `x` is considered to be an angle in radians.

Conditions

Any preconditions and postconditions.

This page titled 5.3: Function Documentation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **5.3: Function Documentation** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

5.4: Naming Functions

There are a few “gotchas” that come up when you start defining functions. The first is that the “real” name of your function is determined by the file name, *not* by the name you put in the function signature. As a matter of style, you should make sure that they are always the same, but if you make a mistake, or if you change the name of a function, it’s easy to get confused.

In the spirit of making errors on purpose, edit *myfunc.m* and change the name of the function from `myfunc` to `something_else` , like this:

```
function res = something_else (x)
    s = sin(x);
    c = cos(x);
    res = abs(s) + abs(c);
end
```

Now call the function from the Command Window, like this:

```
>> y = myfunc(1)
y = 1.3818
```

The function is still called `myfunc` , because that’s the name of the file. If you try to call it like this:

```
>> y = something_else(1)
Undefined function or variable 'something_else'.
```

It doesn’t work. The name of the file is what matters; the name of the function is ignored.

The second “gotcha” is that the name of the file can’t have spaces. For example, if you rename the file to *my func.m* and try to run it, you get:

```
>> y = my func(1)
 y = my func(1)
        |
Error: Unexpected MATLAB expression.
```

This fails because MATLAB thinks `my` and `func` are two different variable names.

The third “gotcha” is that your function names can collide with built-in MATLAB functions. For example, if you create an M-file named *sum.m* and then call `sum` , MATLAB might call your new function, not the built-in version! Which one actually gets called depends on the order of the directories in the search path and (in some cases) on the arguments. As an example, put the following code in a file named *sum.m*:

```
function res = sum(x)
   res = 7;
end
```

And then try this:

```
>> sum(1:3)

ans = 6

>> sum

ans = 7
```

In the first case MATLAB used the built-in function; in the second case it ran your function! This kind of interaction can be very confusing. Before you create a new function, check to see if there is already a MATLAB function with the same name. If there is, choose another name!

This page titled 5.4: Naming Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **5.4: Naming Functions** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

5.4.1: Function Names and MATLAB Keywords

User defined MATLAB function names follow the same rules as variable names:

The function name must start with a letter.

It may have numbers and/or an underscore "_"

It cannot have a space, a hyphen, parentheses, a period, nor another symbol.

The function filename and the name on the function declaration line (a.k.a. signature) inside the file should be the same. See this link: https://www.mathworks.com/help/matlab/ref/function.html

Do not give a user-defined function the same name as a built-in function. That causes a "name collision". You can check the name in the command window like this:

which sum

Both MATLAB and Octave will tell you that this is a built-in function and will display the path to the library which contains this function.

Suppose you are considering the name "my_super_function" for a function you want towrite. When you type

which my_super_function

MATLAB will tell you that this function is "not found", so it is OK to use "my_super_function" a function name.

Octave will not display a message, but will display the command prompt on a new line. This indicates there is no such such command to report.

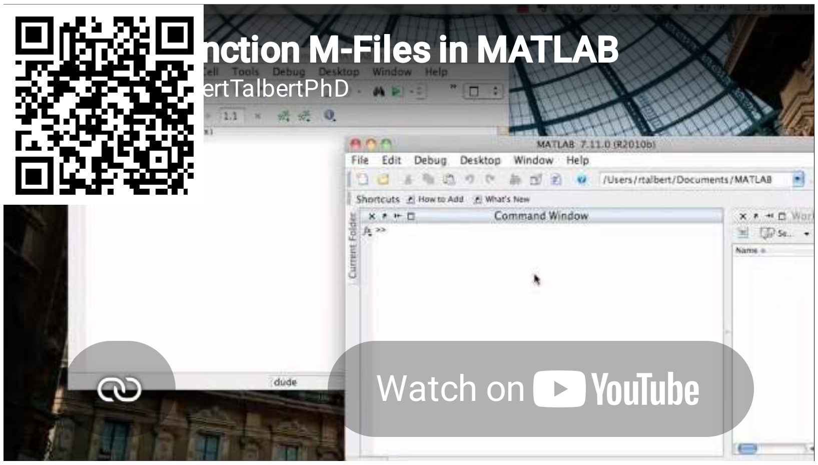
If you make a mistake and give your function the name of a built-in function, you can undo this by renaming your function, then enter this command:

clear functions

MATLAB has keywords that are reserved for its own use.
You cannot use these words as names of variables or functions.
Entering "iskeyword" gives a list of MATLAB's keywords in alphabetical order:

'break'
'case'
'catch'
'classdef'
'continue'
'else'
'elseif'
'end'
'for'
'function'
'global'
'if'
'otherwise'
'parfor'
'persistent'
'return'
'spmd'
'switch'
'try'
'while'

This video gives a clear explanation of defining a MATLAB function:


nction M-Files in MATLAB
ertTalbertPhD
MATLAB 7.11.0 (R2010b)
File Edit Debug Desktop Window Help
/Users/rtalbert/Documents/MATLAB
Shortcuts How to Add What's New
Command Window
Current Folder
dude
Watch on YouTube

Section 10.2 reminds us not to create a variable nor a our own function with the same name as a MATLAB function.

That is, don't create a variable nor a function with a name such as sin, sqrt,

---

This page titled 5.4.1: Function Names and MATLAB Keywords is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.5: Functions With Multiple Input Variables

Functions can take more than one input variable. For example, the following function in Listing 5.2 takes two input variables, `a` and `b`:

Listing 5.2: A function that computes the sum of squares of two numbers

```
function res = sum_squares(a, b)
    res = a^2 + b^2;
end
```

This function computes the sum of squares of two numbers, `a` and `b`.

If we call it from the Command Window with arguments 3 and 4, we can confirm that the sum of their squares is 25.

```
>> ss = sum_squares(3, 4)
ss = 25
```

The arguments you provide are assigned to the input variables in order, so in this case 3 is assigned to `a` and 4 is assigned to `b`. MATLAB checks that you provide the right number of arguments; if you provide too few, you get

```
>> ss = sum_squares(3)
Not enough input arguments.

Error in sum_squares (line 4)
    res = a^2 + b^2;
```

This error message might be confusing, because it suggests that the problem is in `sum_squares` rather than in the function call. Keep that in mind when you're debugging.

If you provide too many arguments, you get

```
ss = sum_squares(3, 4, 5)
Error using sum_squares
Too many input arguments.
```

That's a better error message, because it's clear that the problem isn't in the function, it's in the way we're using the function.

This page titled 5.5: Functions With Multiple Input Variables is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **5.5: Multiple Input Variables** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

5.5.1: Multiple Inputs Examples and Exercises

By Carey A. Smith

More information on creating function is at:

https://www.mathworks.com/help/matlab/ref/function.html?s_tid=doc_ta

✓ Example 5.5.1.1 atan2(y,x) is a built-in function with 2 inputs

Syntax: angle = atan2(y, x). It computes the arctangent of (y/x).
Having the 2 inputs, it can tell which quadrant the angle is in.
Thus, atan2(1, -2) = 2.6779 radians, while atan2(-1, 2) = -0.4636 radians.
By contrast, the corresponding function with 1 input is atan().

atan(-1/2) = -0.4636 radians; it can’t distinguish the 2 cases.

Solution

Add example text here.

✓ Example 5.5.1.2 add1(a,b)

Function syntax: result = add1(a,b)

```
% add1.m:
function result = add1(a,b)
% This adds vectors a and b
% The inputs are a and b
% a and b need to be vectors of the same length
% When there are 2 or more inputs, the inputs are separate by commas
% The output is "result"
result = a + b;
end
```

```
Test code: add_test1.m:
% Clear any variables etc.
clear all; close all; format compact; clc;
x = 1:4
y = [0, 1, 10, 100]
z = add1(x,y)
```

Solution

Add example text here.

.

? Exercise 5.5.1.1 Rocket Height

A student launches a model rocket that has no parachute.

a. (5 pts) Create a function m-file called rocket_height_your_name.m

(Don't use height.m--that's a built-in function.)

The function has 3 input variables (t, h1, v1) and 1 output variable, h.

b. (3 pts) Write these comment lines at the beginning of the function file that describe what the function does.

% This function computes the height of a model rocket vs. time after the engine burns out.

% t % (s) time vector, starting at engine burn out.

% h1 % (m) height at engine burn out.

% v1 % (m/s) velocity at engine burn out.

h % (m) height as a function of time (output)

When the engine burns out, the rocket is at a height of h1 meters with an upward velocity of v1.

At this point, gravity is the only force. (The rocket is aerodynamic, so ignore air friction.)

The rocket will continue to move up for a while, then come down.(Note the variable name, h, must be a different than the function name, height)

c. (3 pts) Inside the rocket_height_your_name.m function file, do the following:

c.1 (1 pt) Set a local variable: g = 9.8 % m/s^2

c.2 (2 pts) Compute the formula for the height of the rocket as a function of time:

```
h = h1 + v1*t + (-1/2)*g*t.^2; %(m) height
```

d. (5 pts) Create a test file that does the following:

d.1 It defines these variables:

```
t = 0 : 0.1 : 6.3; % (s) time vector
```

```
h1 = 100 % (m)
```

```
v1 = 15 % (m/s)
```

d.2 It calls the height function

```
h_out = rocket_height(t,h1, v1);
```

d.3 It opens a new figure window.

d.4 It plots h_out vs. t. (t is the x-axis. h_out is the y-axis)

d.5 It turns the grid on and labels the x and y axes.

Answer

Add texts here. Do not delete this text first.

.

Exercise 5.5.1.2 rectangle_draw

Write a function rectangle_draw_YourName. It has 3 inputs and no output variables, but it does create a plot.

% The function's declaration (1st line) is:

function [ ] = rectangle_draw(width, height, color)

It has 3 input variables:
% width (x direction)

% height (y direction)

% color: Can be any one these: 'b', 'c', 'g', 'k', 'r', or 'y'

% The [ ] in the function's declaration indicates that it has no output variable. It draws a figure instead of computing an output value.

Part A. The function will draw individual 4 lines to create the rectangle.

Inside the function, open a figure and set hold on with these 2 lines of code, so that all the lines will be plotted on 1 plot:

```
figure

hold on
```

Then draw 4 line segments to create a rectangle. Think of the base paths on a baseball field. The algorithm is as follows.

```
%% The 1st line segment is like going from home plate to 1st base.

% The code to draw this line needs to specify the starting and ending x and y coordinates for each line segment.

% This line is horizontal, so the y coordinates of both points are the same.

% Home to 1st base:
x1 = [0, width];
y1 = [0, 0];
plot(x1, y1, color, 'LineWidth', 10)

% Note that the x and y vectors have only 2 values.

%% The 2nd line segment is like going from 1st to 2nd base

% This line is vertical, so the x coordinates of both points are the same.

%% 1st base to 2nd base:
x2 = [width, width];
y2 = [0,     height];
plot(x2, y2, color, 'LineWidth', 10)

%% The 3rd line segment is like going from 2nd to 3rd base

% Try to figure this out.

%% The 4th line segment is like going from 3rd base to home plate

% Try to figure this out.
```

Part B.

```
%% Write a test file that tests your function with these inputs:

width = 40;

height = 30;

color = your choice of 'b', 'c', 'g', 'k', 'r', or 'y'
```

In your test file after calling your Rectangle_draw function, add these lines to provide a 10% white-space margin around your rectangle.

```
  xlim([-0.1*width, 1.1*width])
  ylim([-0.1*height, 1.1*height])
```

These functions adjust the x and y axis of the plot, but don't change the rectangle.

.

Partial Answer

The code to draw the 1st 2 lines is given above. The code to drw the 3rd is:

```
%% 2nd base to 3rd base:
x3 = [width,  0];
y3 = [height, height];
plot(x3,y3,color,'LineWidth',10)
```

> .

.

This page titled 5.5.1: Multiple Inputs Examples and Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.5.2: Mutiple Inputs Example- Ideal Gas law (Lambert)

By Adam Lambert, 8.3 Practical Functions

In most cases we're going to put more than one or two lines of code in a function. Typically, we isolate portions of code that perform a specific task and then package them in a function which is built to be as adaptive as possible.

Consider the Ideal Gas Law.

PV = nRT

In earlier chapters we plotted pressure vs: volume for several different temperatures. This nested calculation is a great time to use a function. We can create a function which will accept n,R,T, and array for V as input variables and then return the pressure array.

First, we create the function in a new m-file. Remember that the name of the function and the name of the m-file must match. I'm going to name my function ideal_gas and save the file as ideal_gas.m. The function will accept 4 values as inputs and return one value. The output value and one of the input values will be arrays, but MATLAB will figure that out automatically. Inside the function, we will loop through the volume array and calculate each element of the pressure array.

It is good practice to give a clear description at the top of any non-trivial functions. In some cases it may be appropriate to include more detailed instructions.

```
function P = ideal_gas(n,R,T,V)
% This function uses the ideal gas
% law to calculate pressure over a
% given volume array.
% Pay careful attention to units.
% Variables
% n: number of mols
% R: universal gas constant
% T: temperature
% V: volume (array)
% P: pressure (array)
% calculation
for i = 1:length(V)
    P(i) = (n*R*T)/V(i);
end
end
```

Notice that a single line of code is used to calculate the P array. The for-loop and all of the indexing has been packaged into the ideal gas function. The code is tucked away in a separate m-le, so we don't have to worry about breaking it with accidental editing. Also, the single line of code is much easier for the user the interpret.

Because the function Workspace and the main Workspace are completely separate, we can use the same variable names in both the testing script and the function. This improves readability and interpretability. However, because the function only accepts values and not the variables themselves, it is important to remember the order that is required when we call the function. The function will always take the first value and assign it to the variable n. The second will always be assigned to the variable R. If we pass the values in a different order, then MATLAB will perform the calculation with the values assigned to the wrong variable. This is one reason why it is good practice to provide clear instructions at the top of a function, including variable definitions.

We can write a short loop to plot isotherms like we made in Chapter 6. This time our script will be much simpler and easy to read.

```
% This script utilizes the
% ideal_gas function to plot
% ideal gas law isotherms.
clear all; close all; clc;
% number of mols
n = 1;
% gas constant (J/(mol*K))
R = 8.314;
% temperature (K)
T = 300:100:700;
% volume (mˆ3)
V = [0.01:0.01:0.1];
% function call
% plot inside loop
for i = 1:length(T)
P = ideal_gas(n,R,T(i),V);
plot(V,P)
hold on;
end
% format plot
title('Ideal Gas Law Isotherms')
xlabel('Volume (mˆ3)')
ylabel('Pressure (Pa)')

legend('T= 300','T= 400','T= 500','T= 600','T= 700')
```

Rather than interpreting a nested loop, we can easily see that we are looping through the temperature values and calculating the pressure-volume curve for each one.

.

This page titled 5.5.2: Mutiple Inputs Example- Ideal Gas law (Lambert) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.6: Functions with Multiple Outputs

Tags recommended by the template: article:topic

By Carey A. Smith

Matlab functions can have more than one output variable. For example, max(a) is a built-in function. It can be used as follows:

```
a = [0, 2, -1, 4.2, 2.1, 0.7]
```

With 1 output argument, it returns the maximum value:

```
a_max = max(a)

a_max = 4.2
```

With 2 output values, it returns the maximum value and the index in a of the maximum value:

```
[a_max, idx_max] = max(a)

a_max = 4.2 % The maximum value

idx_max = 4 % The maximum is the 4th element of a
```

When 2 or more output values, they are listed in square brackets [ ] as a vector.

✓ Example 5.6.1 A function with 2 outputs that computes statistics of a vector

```
function [x_mean, x_std_dev] = stats(x)

% Compute the mean and standard deviation of vector x

x_mean = mean(x); % mean = average

x_std_dev = std(x); % standard deviation

end
```

The first line of the function specifies the 2 output values in square brackets, which is consistent with how it is called:

```
[a_mean, a_std_dev] = stats(a)
```

Test it for a = [0, 2, -1, 4.2, 2.1, 0.7]
Note that “a” is the name of the vector in the main workspace, while "x" is the name of the local variable inside the function.

Also, the variables inside the function are x_mean and x_std_dev, so that they are different than the names of the built-in function's variables mena() and std().

Solution

The outputs are:

```
a_mean = 1.3333

a_std_dev = 1.837
```

.

? Exercise 5.6.1 sum_diff

Write a function called sum_diff.m that has 2 inputs, x and y, and 2 outputs:.

xy_sum (=the sum of vectors x and y)

xy_diff (= the difference of vectors x and y).

Write a 2nd m-file that tests your function with these vectors:

`x = -2:2:6;`

`y = 1:1:5;`

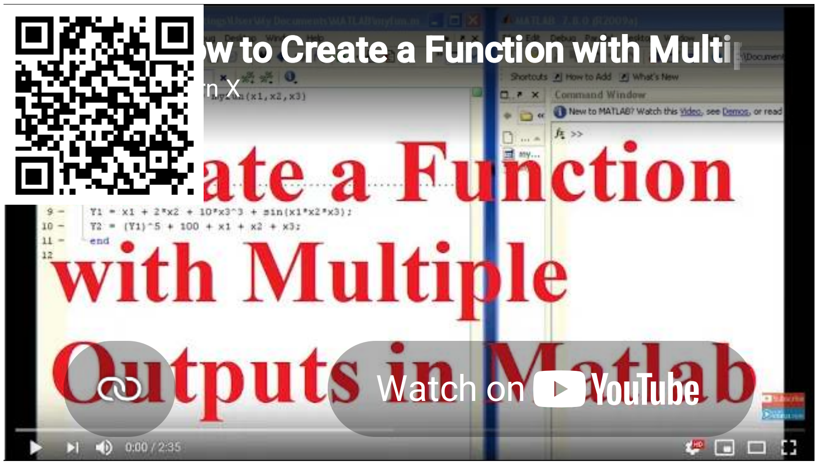
The test code needs to save both outputs with code like this:

[xy_sum, xy_diff] = sum_diff_YourName(x, y)

Answer

Work out the solution.

This video provides additional information:



```
Y1 = x1 + 2*x2 + 10*x3^3 + sin(x1*x2*x3);
Y2 = (Y1)^5 + 100 + x1 + x2 + x3;
end
```

.

? Exercise $5.6.2$ Water Quality

%% The following data is from the Camroasa Water District's 2020 Water Quality report.
% (1pt) Define this vector of chloride levels from imported water and from wells:
Chloride = [52 113 122 45 83 133];

% Read the help for the max() function by typing this:
help max

% (2 pts) Use this form to determine both the maximum value and the index of the maximum:
[Mx, Idx_max] = max(Chloride) % Returns the index corresponding to the maximum value

% (1 pt) Verify that the index is correct with this line:
Chloride(Idx_max)

%% (3 pts) Then use a similar form to determine the minimum and the index of the minimum. Verify that the index of the minimum is correct.

% The minimum function is min()

Answer

Work out the code for this exercise.

.

This page titled 5.6: Functions with Multiple Outputs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.7: Matlab Functions with No Inputs

Tags recommended by the template: article:topic

By Carey A. Smith

Functions can have outputs, even though they have no input arguments.

These are some built-in Matlab functions with no inputs:

date % Try it

datetime % Try it

The function definition of such a file would be like this:

function [date1] = date()

An interesting Matlab function with no inputs is:

pi

This is used as a constant, but it is an actual function. You could define your own functions for constants such as Avogadro's number, the universal gravitational constant, the speed of light, or a parsec's distance.

✓ Example 5.7.1 draw_ellipse.m

This example is a function (draw to draw a circle or an ellipse.

It has 3 input variables, but no output variable.

A figure should be opened before calling this function.

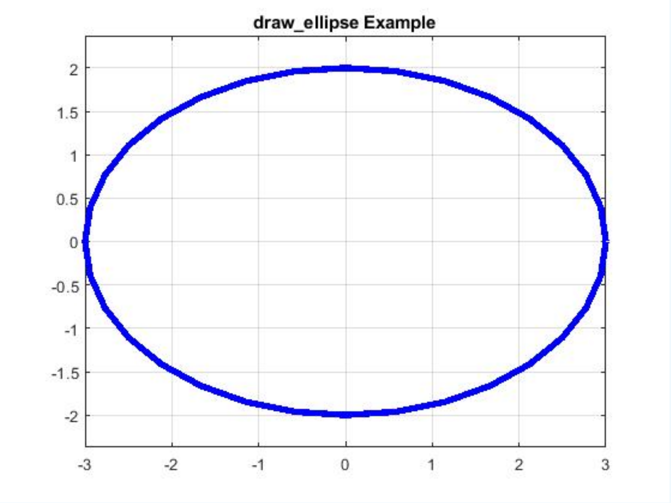
```
function [ ] = draw_ellipse(rx, ry, color)
% This function draws an ellipse or a circle
%  by drawing short lines between 32 point on a circle
% The inputs are:
%  rx = width (x direction)
%  height (y direction)
%  color: One these: 'b', 'c', 'g', 'k', 'r', or 'y'
% There is no returned value.
dth = pi/16;  % (radians) The angle between points
theta = 0: dth : 2*pi;
x1 = rx*cos(theta);
y1 = ry*sin(theta);
plot(x1,y1,color,'LineWidth',4)
end
```

After calling the function, the user should use `axis  equal` so that the plot has the correct shape.

This script (draw_ellipse_example.m) demonstrates the draw_ellipse function

```
figure;
rx = 3
ry = 2;
draw_ellipse(rx, ry, 'b')
axis equal
grid on;
title('draw\_ellipse Example')
```

Solution



.

This page titled 5.7: Matlab Functions with No Inputs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.8: Matlab Functions with No Outputs

Tags recommended by the template: article:topic

By Carey A. Smith

A MATLAB function can have no output variable, although it may display something or do some action. An example is:

clear all -- clears all variables

The user-defined function below draws a star, but does not return a value. Note that in the function declaration line (signature) that empty square brackets are placed where the output value would normally go. The inputs are the size and color of the star.

✓ Example 5.8.1 Function with no outputs: starxy()

```
function [] = starxy(r, color)

% Draw a star of radius r

% color is a character, such as 'r'

dth = (4*pi/5); % delta theta

theta = (pi/2) + [1:1:6]*dth;

x = r*cos(theta);

y = r*sin(theta);

figure;

plot(x, y);

axis equal;

end
```

Test of starxy():

```
starxy(2, 'g') % This draws the green star shown in the figure.
```

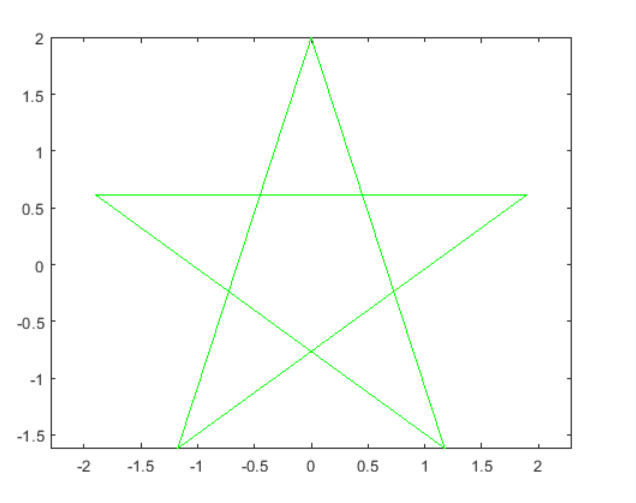


Figure \(\PageIndex{i}\): Function with no outputs: starxy() draws a star.

Solution

Add example text here.

MATLAB and Octave also allow the []= to be omitted from the function declaration when there is no return value. So this declaration also works:

function starxy(r, color)

.

Exercise 5.8.1 Rectangle Draw

Part A. Write a function rectangle_draw_YourName

% Function's 1st line is:

```
function [ ] = rectangle_draw_YourName(width, height, color)
```

```
% It has 3 input variables (a.k.a. arguments):
% width (x direction)
```

```
% height (y direction)
```

```
% color: Can be any one these: 'b', 'c', ''g', 'k', 'r', or 'y'
```

```
% The [ ] in the function's 1st line indicates that it has no output variable. It draws a figure instead of computing an output value.
```

Inside the function, open a figure and turn on hold.

```
figure;
hold on;
```

Then draw 4 line segments to create a rectangle. Think of the base paths on a baseball field.
The 1st line segment is like going from home plate to 1st base.

This line is horizontal, so the y coordinates of both points are the same.

```
% Home to 1st base:
x1 = [0, width];
y1 = [0, 0];
plot(x1,y1,color,'LineWidth',10)
```

The 2nd line segment is like going from 1st to 2nd base.

This line is vertical, so the x coordinates of both points are the same.

```
%% 1st base to 2nd base:
x2 = [width, width];
y2 = [0, height];
plot(x2,y2,color,'LineWidth',10)
```

The 3rd line segment is like going from 2nd to 3rd base. Try to figure this out.

The 4th line segment is like going from 3rd base to home plate. Try to figure this out.

Part B. Write a test file that tests your function with these inputs:

```
width = 40;
```

```
height = 30;
```

```
color = your choice of 'b', 'c', 'g', 'k', 'r', or 'y'
```

```
rectangle_draw_YourName(width, height, color)
```

Answer

```
function [ ] = rectangle_draw(width, height, color)
%% The code opens a figure and draws a rectangle
% The inputs are:
% width (x direction)
% height (y direction)
% color: One these: 'b', 'c', 'g', 'k', 'r', or 'y'
% There is no returned value.
figure;
hold on;

%% It will draw 4 line segments.
% Think of these as the lines between the 4 bases in baseball.
% For each line segment, specify a vector of the first and last x coordinates
% and a vector of the first and last y coordinates.

% Note that the x and y vectors have only 2 values.

%% Home to 1st base:
x1 = [0, width];
y1 = [0, 0];
plot(x1,y1,color,'LineWidth',10)

%% 1st base to 2nd base:
x2 = [width, width];
y2 = [0, height];
plot(x2,y2,color,'LineWidth',10)

%% 2nd base to 3rd base:
x3 = [width, 0];
y3 = [height, height];
plot(x3,y3,color,'LineWidth',10)

%% 3rd base to home
x4 = [0, 0];
y4 = [height, 0];
plot(x4,y4,color,'LineWidth',10)

%% % Set the x and y limits bigger than the rectangle
xlim([-0.1*width, 1.1*width])
ylim([-0.1*height, 1.1*height])
```

.

? Exercise 5.8.1 Trinagle Draw

You can use refer to the rectangle_draw.m function as a pattern for this assignment.

Assignment:

Write a function triangle_draw_YourName

The code opens a figure and draws a right triangle

% The inputs are:
% width
% height

% Option: specify the color as a 3rd input. This is not required.

% The function draws 3 line segments.

% The 1st line segment is horizontal. x goes from 0 to width; y is 0.

% The 2nd line segment is vertical. It goes up by height from the end of the 1st segment.

% The 3rd line segment goes from the end of the 2nd segment back to the strating point (0, 0). This line is diagonal.

% Write a test file that tests your function with these inputs:

width = 40;

height = 30;

Answer

The answer is not given here.

.

This page titled 5.8: Matlab Functions with No Outputs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.9: Matlab Functions with No Inputs and No Outputs

Tags recommended by the template: article:topic

By Carey A. Smith

These built-in Matlab functions have no inputs and no returned variables, although they may display values, change the screen, or change a plot.

colorbar % For 3D plots

lc % Clears the console

The function definition of such a file would be like this:

function [] = clc()

A user could create a function with no inputs and no output variables, but it displays a figure or information. For example, a function called my_contact.m could display a user’s name and contact information:

function [] = my_contact()

Name = 'Harper Lee'

Email = ToKillAMockingbird@google.com

end

This page titled 5.9: Matlab Functions with No Inputs and No Outputs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.10: Functions with variable numbers of inputs or outputs

By Carey A. Smith

Many built-in MATLAB/Octave functions can be used with either a variable number of inputs or a variable number of outputs.

For example the max() function can be used in the following ways:

v = [0, 2, -2, 3, 1]

w = [-3, 3, 0, 1, 11]

v_max = max(v) % This returns the maximum value of the vector.

[v_max, idx_max] = max(v) % This returns both the maximum value of the vector and the index of this maximum value.

vw_max = max(v, w) % This returns a vector of the maximum corresponding values of the 2 vectors.

In order to see how MATLAB functions do this, we can look at the code for the peaks() function. This is a built-in example that creates a 3D plot.

✓ Example 5.10.1 How the peaks() function handles a variable number of inputs and outputs

Try these example to see what happens.

figure;

% 1a. No inputs, no outputspeaks; This creates a 3D plot on a 49x49 grid of x and y points. It also displays the algebraic function that creates the z values.

% 1b. No inputs, 3 outputs

[X,Y,Z] = peaks; % Outputs the 3D data, but does not create the plot.

% 2a. 1 input, no outputs

peaks(17); % Creates a 3D plot on a 17x17 grid of x and y points.

% 2b. 1 input, 3 outputs

[X,Y,Z] = peaks(17); % Outputs the 3D data, but does not create the plot

% 2a. Input 1 vector of axis data values

V = -2: 0.2: 1;
peaks(V); % Creates a 3D plot with x & y axes = V

% 3a. Input a pair of x and y coordinates

Z = peaks(1,2) % Computes the function at the point (X,Y). It does not create a plot.

In the command window, highlight the word peaks, right-click, and open the selection (file). You do not need to understand all the logic here. The important items are the following:

Line of 43: if nargin == 0

nargin is a MATLAB created variable that equals the number of variables input to the function when it was called.

Line 60: if nargout > 1

nargout is a MATLAB created variable that equals the number of variables in the output list when the function was called.

The peaks.m code then uses logic to determine what to compute and plot.

Learning to use of nargin and nargout in your functions is beyond this textbook. All you need to understand is many MATLAB/Octave functions use these varaibles to allow them to be called in multiple ways.

Solution

> Add example text here.

This page titled 5.10: Functions with variable numbers of inputs or outputs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.11: How Functions Work

Let’s review the sequence of steps that occur when you call a function:

1. Before the function starts running, MATLAB creates a new workspace for it.
2. MATLAB evaluates each of the arguments and assigns the resulting values, in order, to the input variables (which live in the *new* workspace).
3. The body of the code executes. Somewhere in the body a value gets assigned to the output variable.
4. The function’s workspace is destroyed; the only thing that remains is the value of the output variable and any side effects the function had (like displaying values).
5. The program resumes from where it left off. The value of the function call is the value of the output variable.

When you’re reading a program and you come to a function call, there are two ways to interpret it. You can think about the mechanism I just described, and follow the execution of the program into the function and back, or you can assume that the function works correctly, and go on to the next statement after the function call.

When you use a built-in function, it’s natural to assume that it works, in part because you don’t usually have access to the code in the body of the function. But when you start writing your own functions, you might find yourself following the “flow of execution.” This can be useful while you are learning, but as you gain experience, you should get more comfortable with the idea of writing a function, testing it to make sure it works, and then forgetting about the details of how it works.

Forgetting about details is called *abstraction*; in the context of functions, abstraction means forgetting about *how* a function works and just assuming (after appropriate testing) that it works. For many people, it takes some time to get comfortable with functions. If you are one of them, you might be tempted to avoid functions, and sometimes you can get by without them.

But experienced programmers use functions extensively, for several good reasons. First, each function has its own workspace, so using functions helps avoid name collisions. Functions also lend themselves to incremental development: you can debug the body of the function first (as a script), then encapsulate it as a function, and then generalize it by adding input variables.

Also, functions allow you to divide a large problem into small pieces, work on the pieces one at a time, and then assemble a complete solution.

Once you have a function working, you can forget about the details of how it works and concentrate on what it does. This process of abstraction is an important tool for managing the complexity of large programs.

This page titled 5.11: How Functions Work is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.9: How Functions Work** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

5.12: Chapter Review

Now that we know about functions, and all the ways they can go wrong, let’s put them to good use. In the next chapter we’ll develop a program that uses several functions to search for Pythagorean triples (and I’ll explain what those are).

Here are a few terms in this chapter you might want to remember.

A *function* is a named sequence of statements stored in an M-file. A function can have one or more *input variables*, which get their values when the function is called, and *output variables*, which return a value from the function to the caller.

The first line of a function definition is its *signature*, which specifies the name of the function, the input variables, and the output variables.

A *silent function* doesn’t display anything, generate a figure, or have any other effect other than returning output values.

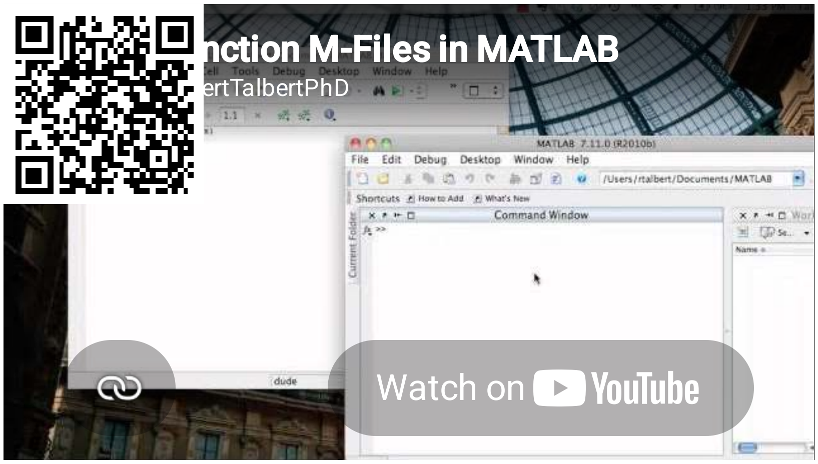
This page titled 5.12: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **5.6: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

5.13: Chapter 5 Function videos English and Espanol

English

Function M-Files in MATLAB-RTalbert video


nction M-Files in MATLAB
ertTalbertPhD
Watch on YouTube

Matlab Function video--MathWorks

https://www.mathworks.com/videos/fun...ons-97413.html

Espanol

OCTAVE - Funciones y Script Files

TAVE - Funciones y Script Files
io Rafael Nolasco Estrada
Watch on YouTube

Funciones - 17 - Octave

nciones - 17 - Octave
siored
tave
17 Funciones
Watch on YouTube

This page titled 5.13: Chapter 5 Function videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

5.14: Exercise

Before you go on, you might want to work on the following exercise.

Exercise 5.1

Write a function called `hypotenuse` that takes two parameters, `a` and `b`, that represent the lengths of two sides of a right triangle. It should assign to `res` the length of the third side of the triangle, given by the formula

$$c=\sqrt{a^2+b^2}$$

This page titled 5.14: Exercise is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith via source content that was edited to the style and standards of the LibreTexts platform.

- **5.7: Exercise** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

CHAPTER OVERVIEW

6: Conditionals Part 1

In this chapter, we'll use functions and a new feature—conditional statements—to search for Pythagorean triples. A Pythagorean triple is a set of integers, like 3, 4, and 5, that are the lengths of the sides of a right triangle. Mathematically, it's a set of integers a, b, and c such that $a^2 + b^2 = c^2$. This example will also demonstrate the *incremental development* process we talked about in Chapter 3.

- 6.1: Relational Operators
  - 6.1.1: Relational Operators List
  - 6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction
- 6.2: if Statement
- 6.3: if, else, elseif
- 6.4: Incremental Development
- 6.5: Logical Functions
- 6.6: Nested for Loops
- 6.7: Putting It Together
- 6.8: Encapsulation and Generalization
- 6.9: Adding a continue Statement
- 6.10: Chapter Review
- 6.11: Chapter 6 if-logic videos English and Espanol
- 6.12: Exercise

This page titled 6: Conditionals Part 1 is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith via source content that was edited to the style and standards of the LibreTexts platform.

6.1: Relational Operators

Suppose we have three variables, `a` , `b` , and `c` , and we want to check whether they form a Pythagorean triple. We can use the equality operator (`==`) to compare two values:

```
>> a = 3;
>> b = 4;
>> c = 5;
>> a^2 + b^2 == c^2

ans = logical 1
```

The result is a *logical* value, which means it's either `1` , which means "true," or `0` , which means "false." Here's an example where the result is false:

```
>> c = 6;
>> a^2 + b^2 == c^2
ans = logical 0
```

It's a common error to use the assignment operator (`=`) instead of the equality operator (`==`). If you do, you get an error:

```
>> a^2 + b^2 = c^2
 a^2 + b^2 = c^2
           |
Error: Incorrect use of '=' operator.
To assign a value to a variable, use '='.
To compare values for equality, use '=='.
```

The equality operator is one of several *relational operators*, so called because they test relations between values. For example, `x < 10` is true (`1`) if the value of `x` is less than `10` or false (`0`) if otherwise. And `x > 0` is true if `x` is greater than `0` .

The other relational operators are `<=` for "less or equal," `>=` for "greater or equal," and `~=` for "not equal."

This page titled 6.1: Relational Operators is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.1: Relational Operators** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.1.1: Relational Operators List

Relational Operators List (from Mathworks help)

==	equal (used to determine equality)	(a == b)
>=	Greater than or equal (To determine greater than or equal to)	(a >= b)
>	Greater than (used to determine greater than)	(a > b)
<=	Less than or equal (used to determine less than or equal to)	(a <= b)
<	Less than (used to determine less than)	(a < b)
~=	"not equal" (used to determine inequality)	(a ~= b)

See also:

https://www.mathworks.com/help/matlab/relational-operators.html

---

This page titled 6.1.1: Relational Operators List is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction

Here are some codes that Octave allows, but which are not compatible in MATLAB.

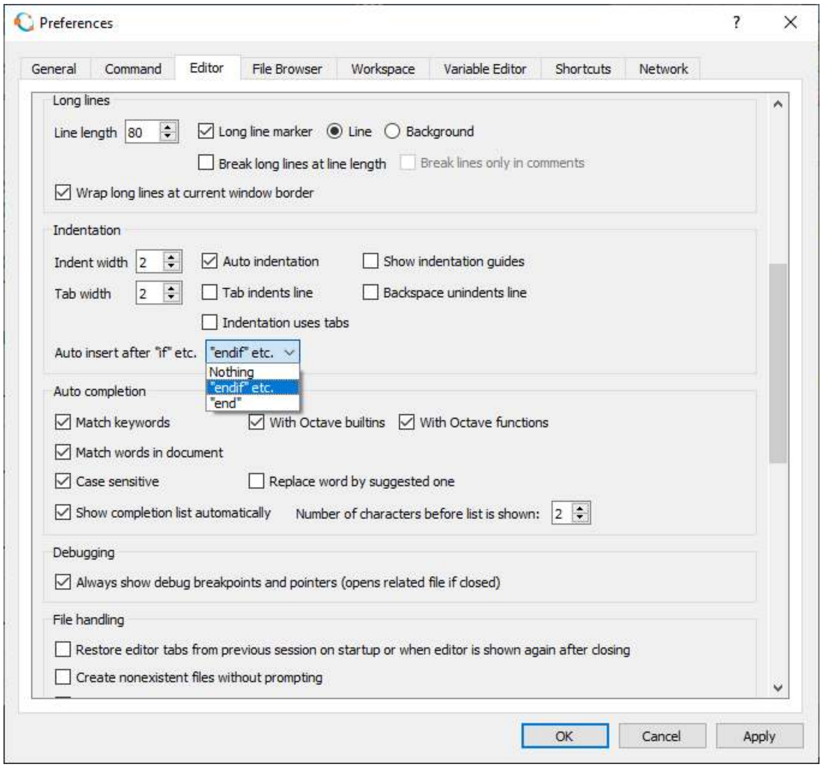
Octave users should use MATLAB compatible code, so that their code can run in either Octave or MATLAB

1. In MATLAB, the not equal symbol is ~=. Octave allows either ~= or !=

2. In MATLAB, a function should end with the keyword "end". Octave allows either "end" or "endfunction". By default, Octave inserts "endfunction".

3. In MATLAB, an if logic block of code must end with the keyword "end". Octave allows either "end" or "endif". By default, Octave inserts "endif".You change Octave's default to “end” by doing the following:

From the main window’s top menu bar, click on “editor”, then click on “Preferences”, which is the last in the list of options.
In the preferences window, change ‘Auto insert after “if” etc.’ from endif to end. Then click Okay.



Long lines
Line length 80 ☑ Long line marker ● Line ○ Background
☐ Break long lines at line length ☐ Break lines only in comments
☑ Wrap long lines at current window border

Indentation
Indent width 2 ☑ Auto indentation ☐ Show indentation guides
Tab width 2 ☐ Tab indents line ☐ Backspace unindents line
☐ Indentation uses tabs
Auto insert after "if" etc. "endif" etc.
Nothing
"endif" etc.
"end"

Auto completion
☑ Match keywords ☑ With Octave builtins ☑ With Octave functions
☑ Match words in document
☑ Case sensitive ☐ Replace word by suggested one
☑ Show completion list automatically Number of characters before list is shown: 2

Debugging
☑ Always show debug breakpoints and pointers (opens related file if closed)

File handling
☐ Restore editor tabs from previous session on startup or when editor is shown again after closing
☐ Create nonexistent files without prompting

.

This page titled 6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

6.2: if Statement

Now suppose that when we find a Pythagorean triple we want to display a message. The `if` statement allows you to check for certain conditions and execute statements if the conditions are met. For example:

```
if a^2 + b^2 == c^2
    disp("Yes, that is a Pythagorean triple.")
end
```

The syntax is similar to a `for` loop. The first line specifies the condition we’re interested in. If the condition is true, MATLAB executes the *body* of the statement, which is the indented sequence of statements between the `if` and the `end` .

MATLAB doesn’t require you to indent the body of an `if` statement, but it makes your code more readable, so you should do it.

If the condition is not satisfied, the statements in the body are not .

Sometimes there are alternative statements to execute when the condition is false. In that case, you can extend the `if` statement with an `else` clause.

The complete version of the previous example might look like this:

```
if a^2 + b^2 == c^2
    disp("Yes, that is a Pythagorean triple.")
else
    disp("No, that is not a Pythagorean triple.")
end
```

Statements like `if` and `for` that contain other statements are called *compound* statements. All compound statements finish with `end` .

This page titled 6.2: if Statement is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.2: if Statement** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.3: if, else, elseif

Most of the following explanations of if, else, elseif, usage are primarily from Donald Dabdub, ConditionalStatements_FlowControl.pdf, http://albeniz.eng.uci.edu/mae10/PDF...lowControl.pdf

One example is by Carey Smith.

Often we want to execute a command only if a certain test condition is satisfied. We use if statements to do this.

6.3.1: Simple if statement

If you want to execute certain commands only when a certain test condition is met, use a simple if statement of the form:

```
if ( test condition )
   commands
end
```

- The "test condition" is any combination of relational and logical operators used to compare different variables or data.
- You don't need the parentheses around the test condition, but it is a good idea to use parenthesis to improve code readability and avoid confusion and mistakes. Always make sure your parenthesis are balanced to prevent errors.
- The "commands" are any commands, statements, or calculations that you have learned in MATLAB.
- An end is required for every opening if, which signifies the end of all commands that depend on the test condition. Anything outside of the if statement (before the opening if or after the closing end) is executed regardless of whether or not the test condition is satisfied – only the commands inside the if statement depend on the test condition.
- The indentation for the statements inside the if is not required but greatly improves the readability of presentation of the code. MATLAB will even automatically do indentation for you – simply select all the text in your M-File and right click for the "Smart Indent" option (CNTRL + I on Windows). Indentation is always recommended.
- Note that MATLAB is case sensitive – do not capitalize any letters in "if" or "end" as doing so will produce errors.

✓ Example 6.3.1 if x, y example

Let's jump right into an example to see how if statements work.
In myfile.m:

```
x = 5.4
y = 10
if (x>4) % if the test condition is true, the following two lines will be executed
    y = y + 1;
    x = 2; % if the test condition is false, these statements will not be executed
end
```

Sample case initial values:

x = 5.4000

y = 10

Final values:

x = 2

y = 11

In the example above, the test condition on the if statement was satisfied (it was true) since the value stored in x was indeed great than 4. Because the test condition was true, the statements inside the if were executed, and the final values of x and y after the if statement were different than the initial values.

Solution

Add example text here.

.

6.3.2: if, else statement

An if, else statement will allow you to execute certain commands if a test condition is true and execute other commands if the test condition is false. The commands inside the else are executed only if the test condition on the if statement is false. If the test condition on the if statement is true, the commands inside the if are executed, and the commands inside the else are skipped. Thus, only the commands inside the if or the commands inside the else will be executed, never both. Let's look at an example of an if, else construct to illustrate.

✓ Example $6.3.2$ if, else example

```
x = 5 % Set x=5 to demonstrate how the if logic works.
if (x < 0) % This is false for x=5,
    % so the following 2 lines of code are not executed
    z = 100;
    beep
else % The following commands are executed when (x < 0) is false.
    z = 200;
    beep
    pause(0.6) % wait 0.6 seconds between beeps)
    beep
end
z % show the value of z
```

Solution

Add example text here.

.

Watch this video (Logical operators in MATLAB, Robert Talbert) before studying the next example:

https://www.youtube.com/watch?v=68CXBpjCBRw

.

6.3.3: if, elseif, else statement

If you have 2 or more separate conditions that you need to check, but only one can be satisfied at a time, you can use an if, elseif statement. In an if, elseif statement, a series of possible conditions are checked in order. The order is important because as soon as one condition is satisfied, the commands

Note that once a test condition is found to be true, the statements inside that test condition are executed and the if, elseif statement is terminated even if later test conditions would be true too. This brings up a few important points to remember when using an if, elseif construct:

- Only one condition on an if, elseif can be true.
- * If you have 2 or more conditions that need to be checked and it is possible for both of them to be true simultaneously, you should use separate if statements rather than an if, elseif.
- When using an if, elseif, be careful with the order in which you enter your test conditions.
- The following practice problem should be solved to illustrate this point is the following (solve this problem before proceeding with the notes)

Exercise 6.3.1 if, elseif, else Grade (Practice Problem)

- Precondition: the user sets the variable score to a value between 0 and 100.
- Use an if, elseif statement to determine the user's equivalent letter grade;
    - A for score >=90
    - B for score < 90 and >=80
    - C for score < 80 and >=70
    - D for score < 70 and >=60
    - F for score < 60
- (Do this without using compound relations; that will come later.)
- Be careful! Putting test conditions in the wrong order will produce incorrect output for the letter grade. Be sure to test your code with an input in each of the grade ranges.
- Including an else on an if, elseif is optional. An else is never required. Remember that it is only executed if all other test conditions are false

Answer

```
% Precondition: Set variable score = to a value between 0 and 100
if (score >= 90)
    grade = 'A'
elseif (score >= 80)
    grade = 'B'
elseif (score >= 70)
    grade = 'C'
elseif (score >= 60)
    grade = 'D'
elseif (score >= 0)
    grade = 'F'
else
    disp('score is not valid') % Perhaps score is a character, not a number
end
```

Example 6.3.4 if, elseif, else name length

By Carey Smith

This script tests the length of a name and determines if it is short, medium, or long.

```
name_in = input('Enter your name: ','s')
% The 's' tells the input() function that the response will be a string.

name_len = length(name_in) % Number of letters in the name
if(name_len <= 4) % Is the name <= 4 letters?
    disp([name_in,' is a short name'])

elseif(name_len <= 7) % The name is > 4 letters; is the name <= 7 letters?
    disp([name_in,' is a medium name'])

else % the name > 7 letters
    disp([name_in,' is a long name'])
end
```

Solution

Add example text here.

.

? Exercise 6.3.2 if_else_drivers_test

Figure out the code for this exercise:

On the California written drivers license test, you need to score at least 30 out of 36 to pass.
Write a script that does the following:
A. (1 pt) Has an precondition (input) of a variable named score.
B. (2 pts) If score is greater than or equal to 30,
then the script displays this message:
'Congratulations, you passed.'
C. (2 pts) Else, if score is greater than 25 (but less than 30),
then the script displays this message:
'You did not pass, but were close.'
D. (2 pts) Else, the script displays this message:
'You need to study a lot before taking the test again.'

This should be tested with values of score in all 3 ranges.

Answer

Add texts here. Do not delete this text first.

.

6.3.4: Nested if statements

You can put if statements within if statements; this is called nesting and if statement. This is typically done if you only want to check a certain condition if another condition has already been satisfied. Thus, in this construct the nested if statement will be checked only when the if statement that it is inside of becomes true. Let's look at an example to illustrate.

✓ Example 6.3.5 Nested if statements score example

The adds + or - to a grade, such as A+ or A-

```
In myfile.m:
% The input() function displays a message, then stores the user's input in the
variable "score"
score = input('Enter your score: ');
```

```
disp('Your grade is:')
if((score<=100) & (score>=90)) % This compound logic is only true if both parts are
true.
    grade(1) = 'A';
    if(score>=90 & score<93) % This "if statement" is inside the previous "if
statement".
        grade(2) = '-'; % the variable grade has 2 characters
    elseif(score>=93 & score<97)
        grade(2) = ' ';
    elseif(score>=97 & score<=100)
        grade(2) = '+';
    end
    disp(grade) % This displays the value of the variable grade
else
    disp('B or worse')
end
```

Solution

Sample output:
>> myfile
Enter your score: 97.5
Your grade is:
A+

In this example, we only check the nested if statement if the score is within the range for an A grade. If the score is not within this range, the statements inside the else are executed and the nested if statement is skipped entirely.

We will execute the next example twice, with different input values to illustrate the behavior of the nested if statement. In the second execution, the nested if statement will be skipped entirely because the score is not greater than 90. Just remember in the nested if construct, the nested if statements are only checked when the outside if statement that they are inside of becomes true.

✓ Example 6.3.6 Nested if logic score and age

In myfile.m:

```
score = input('What was your test score?: ');
age = input('How old are you?: ');
if(score>90)
    if(age<=18)
        disp('You did very well... for a young person')
    elseif(age<=21 & age>18)
        disp('You did pretty well... for a young person')
    else
        disp('You are old')
    end
else
    disp('Your score is too low')
end
```

Sample run #1:
>> myfile
What was your test score?: 91

How old are you?: 14
You did very good... for a young person

Sample run #2:
>> myfile
What was your test score?: 81
How old are you?: 55
Your score is too low

Solution

Add example text here.

,

This video from "SnugglyHappyMathTime" shows a couple of fun examples:

Else, and Elseif Matlab Statement
gglyHappyMathTime
MyNotes.m*
Command Window
MyNo
Wo...
Name
beautif
Watch on YouTube
script
Ln 57
Col 18
Trial Days Remaining: 130

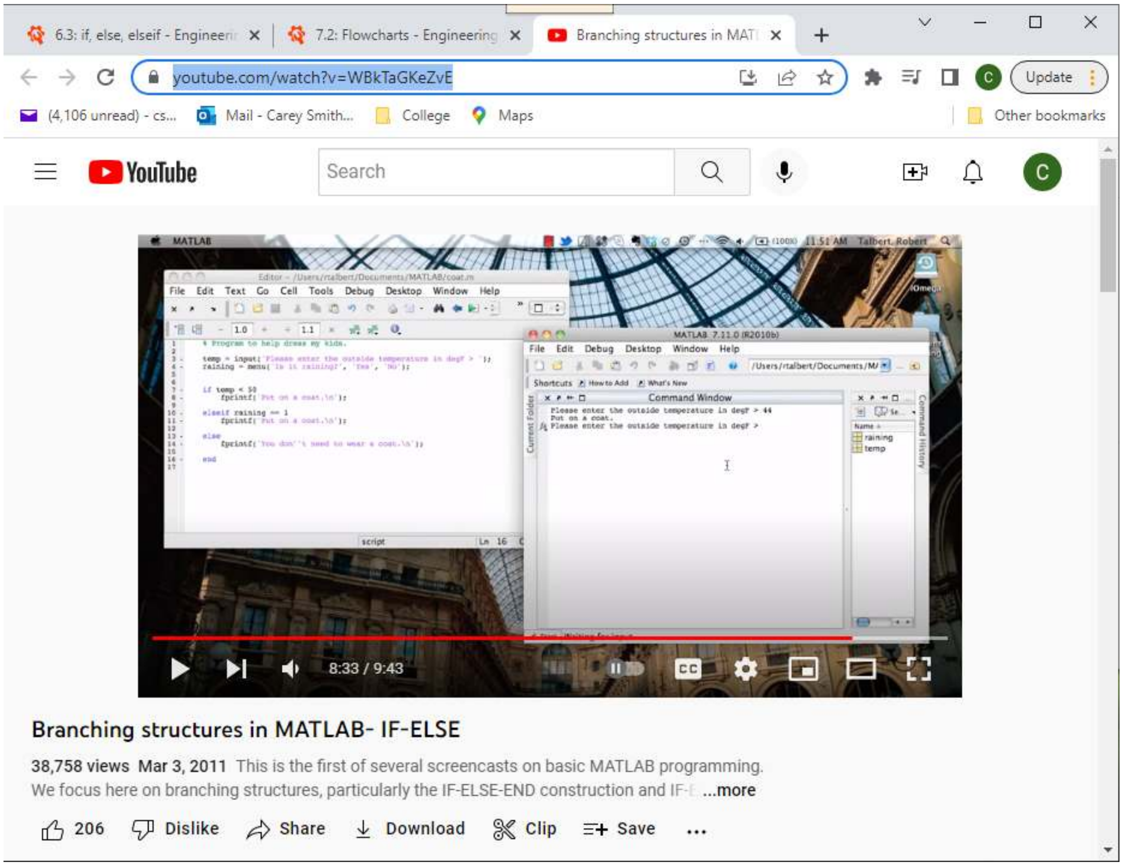
```
r
l = input('how many puppies do you have? ');
iful == 5
        disp('Awesome!')
    else
        disp('onoes!')
    end
```

```
how many puppies do you have? 5
Awesome!
Trial>>
```

This video offers shows nested if-elseif-else-end logic and shows a flowchart of the logic:

Branching structures in MATLAB- IF-ELSE (Robert Talbert)

https://www.youtube.com/watch?v=WBkTaGKeZvE



6.3: if, else, elseif - Engineeri
7.2: Flowcharts - Engineering
Branching structures in MAT
youtube.com/watch?v=WBkTaGKeZvE
Update
(4,106 unread) - cs...
Mail - Carey Smith...
College
Maps
Other bookmarks
YouTube
Search
MATLAB
Editor - /Users/rtalbert/Documents/MATLAB/coat.m
File Edit Text Go Cell Tools Debug Desktop Window Help
MATLAB 7.11.0 (R2010b)
File Edit Debug Desktop Window Help
/Users/rtalbert/Documents/M
Shortcuts How to Add What's New
Command Window
Current Folder
Command History
8:33 / 9:43

```
% Program to help dress my kids.

temp = input('Please enter the outside temperature in degF > ');
raining = menu('Is it raining?', 'Yes', 'NO');


if temp < 50
    fprintf('Put on a coat.\n');

elseif raining == 1
    fprintf('Put on a coat.\n');

else
    fprintf('You don''t need to wear a coat.\n');

end
```

```
Please enter the outside temperature in degF > 44
Put on a coat.
Please enter the outside temperature in degF >
```

Branching structures in MATLAB- IF-ELSE

38,758 views Mar 3, 2011 This is the first of several screencasts on basic MATLAB programming. We focus here on branching structures, particularly the IF-ELSE-END construction and IF-E ...more

206
Dislike
Share
Download
Clip
Save

.

This page titled 6.3: if, else, elseif is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

6.4: Incremental Development

Now that we have relational operators and `if` statements, let's start writing the program.

Here are the steps we will follow to develop the program incrementally:

1. Write a script named *findtriples.m* and start with a loop that enumerates values of `a` and displays them.
2. Write a second loop that enumerates values of `b` and a third loop that enumerates values of `c` .
3. Use the if statement from the previous section to check whether `a` , `b` , and `c` form a Pythagorean triple.
4. Display the values that pass the test.
5. Transform the script into a function and make it take an input variable that specifies the range to search.

Along the way, we'll optimize the program to eliminate unnecessary work.

This page titled 6.4: Incremental Development is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.3: Incremental Development** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.5: Logical Functions

This is a modified version of Allen B. Downey's section. It was modified by Carey Smith to use parentheses for clarity and to remove possible ambiguity.

The first step is to create a *logical function*, which is a function that returns a logical value. The following function takes three input variables, `a` , `b` , and `c` , and returns true (`1`) if they form a Pythagorean triple and false (`0`) otherwise.

```
function res = is_pythagorean(a, b, c)
    if ( (a^2 + b^2) == c^2)
        res = 1;
    else
        res = 0;
    end
end
```

We can use this function like so:

```
>> is_pythagorean(3, 4, 5)
ans = 1
```

Put this function in a file called *is_pythagorean.m*, so we can use it as part of our program.

This page titled 6.5: Logical Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith via source content that was edited to the style and standards of the LibreTexts platform.

- **6.4: Logical Functions** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.6: Nested for Loops

The next step is to write loops that enumerate different values of `a` , `b` , and `c` . Create a new file called *find_triples.m* where we'll develop the rest of the program.

We'll start with a loop for `a` :

```
for a=1:3
    a
end
```

It might seem silly to start with such a simple program, but this is an essential element of incremental development: start simple and test as you go.

The output is as expected.

```
1
2
3
```

Now we'll add a second loop for `b` . It might be tempting to write something like this:

```
for a=1:3
    disp(a)
end
for b=1:4
    disp(b)
end
```

But that loops through the values of `a` and then loops through the values of `b` , and that's not what we want.

Instead, we want to consider every possible pair of values, like this:

```
for a=1:3
    for b=1:4
        disp([a,b])
    end
end
```

Now one loop is inside the other. The inner loop gets executed three times, once for each value of `a` , so here's what the output looks like (I've adjusted the spacing to make the structure clear):

```
>> find_triples
     1     1
     1     2
     1     3
     1     4
     2     1
     2     2
     2     3
     2     4
     3     1
     3     2
     3     3
     3     4
```

The left column shows the values of `a` and the right column shows the values of `b`.

The next step is to search for values of `c` that might make a Pythagorean triple. The largest possible value for `c` is `a + b`, because otherwise we couldn't form a triangle (see https://greenteapress.com/matlab/triangle).

```
for a=1:3
    for b=1:4
        for c=1:a+b
            disp([a,b,c])
        end
    end
end
```

After each small change, run the program again and check the output.

This page titled 6.6: Nested for Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.5: Nested Loops** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.7: Putting It Together

Now instead of displaying all of the triples, we'll add an `if` statement and display only Pythagorean triples:

```
for a=1:3
    for b=1:4
        for c=1:a+b
            if is_pythagorean(a, b, c)
                disp([a,b,c])
            end
        end
    end
end
```

The result is just one triple:

```
>> find_triples
     3     4     5
```

You might notice that we're wasting some effort here. After checking the case when `a` is 1 and `b` is 2, there's no point in checking the case when `a` is 2 and `b` is 1. We can save the extra work by adjusting the range of `b`:

```
for b=a:4
```

We can save even more work by adjusting the range of `c`:

```
for c=b:a+b
```

Here's the final version:

```
for a=1:3
    for b=a:4
        for c=b:a+b
            if is_pythagorean(a, b, c)
                disp([a,b,c])
            end
        end
    end
end
```

This page titled 6.7: Putting It Together is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.6: Putting It Together** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.8: Encapsulation and Generalization

As a script, this program has the side effect of assigning values to `a`, `b`, and `c`, which would be bad if any of those names were in use. By wrapping the code in a function, we can avoid name collisions; this process is called *encapsulation* because it isolates this program from the workspace.

The first draft of the function takes no input variables:

```
function res = find_triples()
    for a=1:3
        for b=a:4
            for c=b:a+b
                if is_pythagorean(a, b, c)
                    disp([a,b,c])
                end
            end
        end
    end
end
```

The empty parentheses in the signature are not necessary, but they make it apparent that there are no input variables. Similarly, it's a good idea when calling the new function to use parentheses as a reminder that it's a function, not a script:

```
>> find_triples()
```

The output variable isn't necessary, either; it never gets assigned a value. But I put it there as a matter of habit and so my function signatures all have the same structure.

The next step is to generalize this function by adding input variables. The natural generalization is to replace the constant values `3` and `4` with a variable so we can search an arbitrarily large range of values.

```
function res = find_triples(n)
    for a=1:n
        for b=a:n
            for c=b:a+b
                if is_pythagorean(a, b, c)
                    disp([a,b,c])
                end
            end
        end
    end
end
```

Here are the results for the range from 1 to 15:

```
>> find_triples(15)
     3     4     5
     5    12    13
     6     8    10
     8    15    17
     9    12    15
```

The triples $5, 12, 13$ and $8, 15, 17$ are new, but the others are just multiples of the $3, 4, 5$ triangle.

This page titled 6.8: Encapsulation and Generalization is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.7: Encapsulation and Generalization** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.9: Adding a continue Statement

As a final improvement, let's modify the function so it only displays the "lowest" of each Pythagorean triple, and not the multiples.

The simplest way to eliminate the multiples is to check whether `a` and `b` share a common factor. If they do, dividing both by the common factor yields a smaller, similar triangle that has already been checked.

MATLAB provides a `gcd` function that computes the greatest common divisor of two numbers. If `gcd(a,b)` is greater than 1, `a` and `b` share a common factor and we can use the `continue` statement to skip to the next pair. Listing 6.1 contains the final version of this function:

Listing 6.1: Our final Pythagorean triples function

```
function res = find_triples(n)
    for a=1:n
        for b=a:n
            for c=b:a+b
                if gcd(a,b) > 1
                    continue
                end
                if is_pythagorean(a, b, c)
                    disp([a,b,c])
                end
            end
        end
    end
end
```

The `continue` statement causes the program to end the current iteration immediately, jump to the top of the loop, and "continue" with the next iteration.

In this case, since there are three loops, it might not be obvious which loop to jump to, but the rule is to jump to the innermost loop (which is what we want).

Here are the results with `n = 40` :

```
>> find_triples(40)
     3     4     5
     5    12    13
     7    24    25
     8    15    17
     9    40    41
    12    35    37
    20    21    29
```

This page titled 6.9: Adding a continue Statement is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.8: Adding a continue Statement** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.10: Chapter Review

In this chapter, we encountered relational operators and `if` statements, and we used them to develop a program that searches for Pythagorean triples. We wrote a *logical function*, which is a function that returns a logical value (`1` for “true” or `0` for “false”).

We also saw an example of *incremental development*, or developing programs gradually, adding just a few lines of code at a time and testing as you go. If you develop programs this way, you will have fewer bugs and you will find them more quickly.

This chapter defined two new terms: *encapsulation* is the process of wrapping part of a program in a function in order to limit interactions (including name collisions) between the function and the rest of the program; *abstraction* is the process of ignoring the details of how a function works in order to focus on a simpler model of what the function does.

The next chapter introduces a new tool, called `fzero` , that we’ll use to solve nonlinear equations.

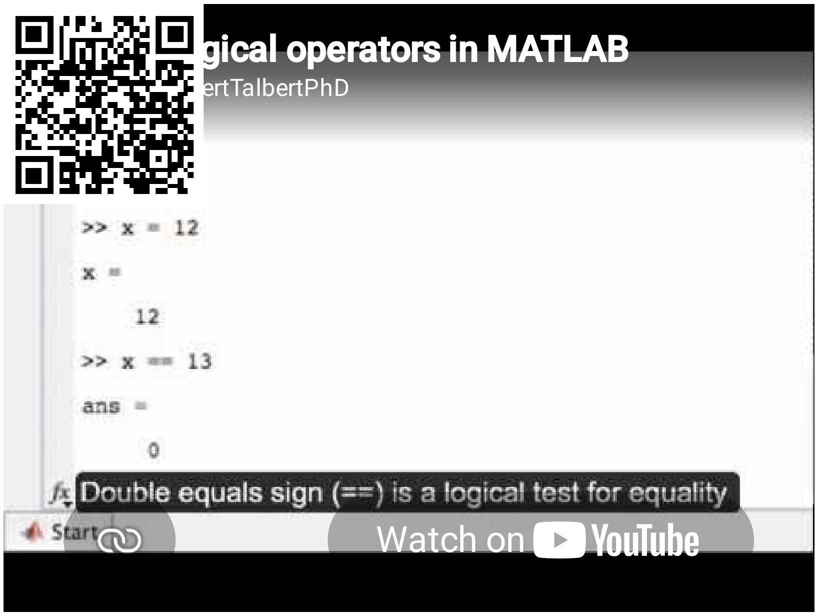
This page titled 6.10: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.10: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

6.11: Chapter 6 if-logic videos English and Espanol

English

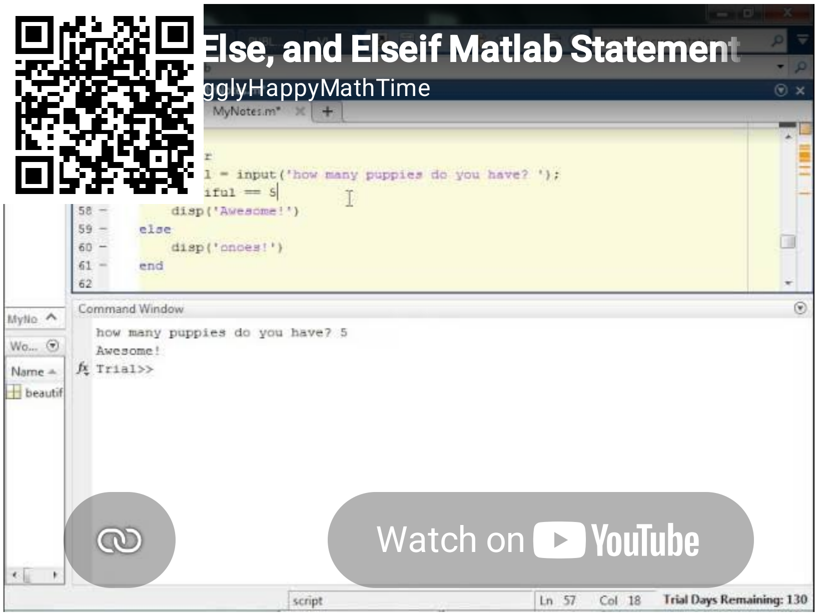
Matlab Logical Operators video, RTalbert


gical operators in MATLAB
ertTalbertPhD
Start
Watch on YouTube

```
>> x = 12

x =

    12

>> x == 13

ans =

     0
```

Double equals sign (==) is a logical test for equality

if-else-elseif video, SnugglyMath


Else, and Elseif Matlab Statement
gglyHappyMathTime
MyNotes.m*
Command Window
MyNo
Wo...
Name
beautif
Watch on YouTube
script
Ln 57
Col 18
Trial Days Remaining: 130

```
r
l = input('how many puppies do you have? ');
iful == 5
        disp('Awesome!')
    else
        disp('onoes!')
    end
```

```
how many puppies do you have? 5
Awesome!
Trial>>
```

Branching Structures in MATLAB: If-Else Video RTalbert

anching structures in MATLAB- IF
ertTalbertPhD
Watch on YouTube

Espanol

Operadores lógicos y relacionales - 10 - Octave

eradores lógicos y relacionales - 1
siored
Operadores
Relacionales y
lógicos
Watch on YouTube

if, else, elseif - 11 - Octave

else, elseif - 11 - Octave
siored
11 if
else, elseif
Watch on YouTube

This page titled 6.11: Chapter 6 if-logic videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

6.12: Exercise

Before you go on, you might want to work on the following exercise.

Exercise 6.1

There is an interesting connection between Fibonacci numbers and Pythagorean triples. If F is a Fibonacci sequence, then

$$\left(F_iF_{i+3},\ 2F_{i+1}F_{i+2},\ F_{i+1}^2+F_{i+2}^2\right)$$

is a Pythagorean triple, for all $i \geq 1$.

Write a function named `fib_triple` that takes `n` as an input variable, computes the first `n` Fibonacci numbers, stores them in a vector, and checks whether this formula produces Pythagorean triples for numbers in the .

This page titled 6.12: Exercise is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **6.11: Exercise** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

CHAPTER OVERVIEW

7: Conditionals Part 2- Series, Switch Logic, Error Checking, and While Loops

7.1: if, elseif, else with Error Checking
7.2: Flowcharts
7.3: The input() Function
7.4: Compound Logical Operators
  7.4.1: Compound if, elseif Logic Examples
7.5: Working with Conditions
7.6: Shortcut and, or, Logical Operations
7.7: for loops with break Logic
7.8: 'while' Loops
7.9: 'switch' Logic
7.10: The find() function
7.11: Chapter 7 Conditionals, Part 2, Videos English and Espanol

.

This page titled 7: Conditionals Part 2- Series, Switch Logic, Error Checking, and While Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

7.1: if, elseif, else with Error Checking

By Carey A. Smith

This section gives more if, elseif, else, examples. It also shows how to use the error() function.

✓ Example 7.1.1 add2() function with error checking

Previously we saw the add1() function, which adds 2 vectors. add1() assumed that the 2 vectors were the same length, but did not check that.

Here we create add2(), which checks that the 2 vectors are the same length.

add2.m:

```
function result = add2(a,b)
% This adds vectors a and b
% The inputs are a and b
% It first checks to see that a and b are the same length
% If not, it creates an error message exits the function.
% The output is "result"
a_len = length(a)
b_len = length(b)
if(a_len == b_len)
    result = a + b;
else % else, the vectors are different lengths, so it creates an error message
    result = 0
    error('length(a) must = length(b)')

     % The error() function displays the message plus the function name and line
where the error occured.

    % Then error() cause the program to stop
end
```

This test script demonstrates normal runs with both add1() and add2() and runs with inputs that are not the same length.

```
add2_test.m

%% Test add functions
% (1 pt) Clear any variables etc.
clear all; close all; format compact; clc;

%% First, test add1.m and add2.m with 2 vectors that are the same length
x = 1:4
y = [0, 1, 10, 100]
z = add1(x,y)
z = add2(x,y)

%% Second, test add1.m and add2.m with 2 vectors that are not the same length
w = [1, 2]
y = [0, 1, 10, 100]
z = add1(w,y)
z = add2(w,y)

The error() function in add2.m displays this message:
```

```
Error using add2 (line 14)
length(a) must = length(b)
```

Solution

Add example text here.

.

Exercise 7.1.1 Area of a Square

(1 pt) Write a function that computes the area of a square that expects a scalar input, not a vector.

The output will be the variable "sq_area"
(1 pt) Use the input function to get the length of a side with this function:
s = input('Enter the length of one side of a square: ');

(2 pts) Check that s is a single number.
% Use s_len = length(s) to find the length of s
% If s_len is not 1, then display s and generate this error:
error('Area_of_square.m: the input must be a single number') % Note the space after the colon
% The error function causes the program to stop

% (2 pts) Check that s is >= 0 (The side can't be < 0)
% If s is negative, then display s and generate this error:
error('Area_of_square.m: the input is < 0')
% The error function causes the program to stop

% (2 pts) Compute sq_area = the area of the square. (Don't call this variable "area", because that is the name of a plotting function.)

% Test it with these input values:
s = 3
s = -1
s = 1:3

Answer

See the attached file Area_of_square.m

7.1.1: Homework:

Exercise 7.1.2 Area of a Circle

% (1 pt) Write a function that computes the area of a circle.
% (1 pt) Use the input function to get the radius with this function:
r = input('Enter the radius of a circle: '); % Note the space after the colon
% The input() function displays the message, then reads a number and stores it in the variable r

% (2 pts) Check that r is a single number.
% Use r_len = length(r) to find the length of r
% If r_len is not 1, then display r and generate this error:
error('Area_of_circle.m: the input must be a single number')
% (2 pts) Check that r is >= 0
% If r is negative, then display r and generate this error:
error('Area_of_circle.m: the input is < 0')
% (2 pts) Compute the area of the circle.

> % Test your function with these input values:
> r = 3
> r = -1
> r = 1:3
>
> Since these are entered via the input() function, you do not need a test file.
>
> **Answer**
>
> Add texts here. Do not delete this text first.

.

7.1.2: Error checking with try, catch keywords

MATLAB and Octave have an advanced feature called try, catch that can be used in place the the error() function.

It allows an error to be caught, the error message to be displayed, and the run to exit more gracefully.

If you are interested, you can learn about try, catch in MATLAB and Octave at these links:

https://www.mathworks.com/help/matlab/ref/try.html

https://docs.octave.org/interpreter/The-try-Statement.html

.

This page titled 7.1: if, elseif, else with Error Checking is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

7.2: Flowcharts

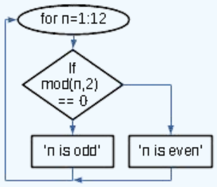
By Carey A. Smith

A flowchart is a graphical representation of code logic. It is especially helpful to understand the logic of code with for loops and if-else logic.

An example is given here. More information can be found at:

https://en.wikipedia.org/wiki/Flowchart [en.Wikipedia.org]

✓ Example 7.2.1 Odd-Even Flow Chart



Try to code this up

Solution

The corresponding code is:

```
for n = 1:12
    if(mod(n,2) == 0)
        disp([num2str(n),' is even'])
    else
        disp([num2str(n),' is odd'])
    end
end
```

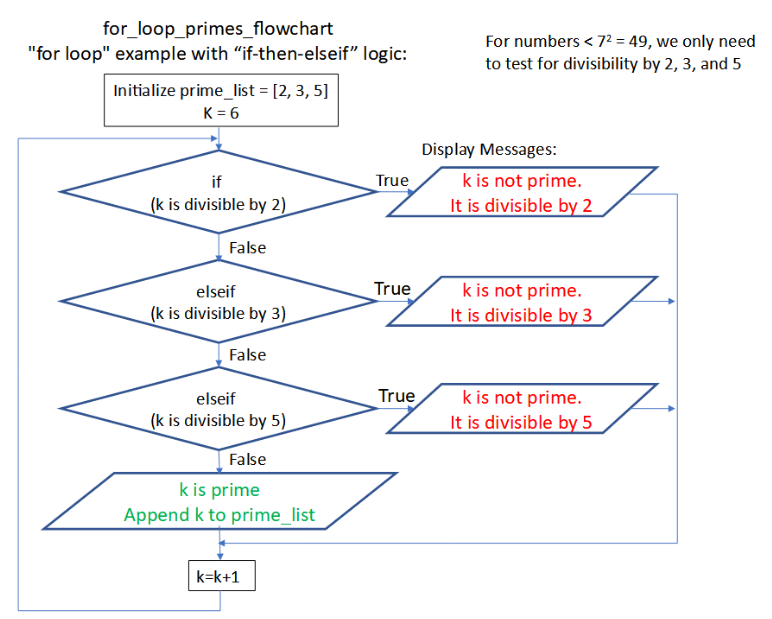
.

✓ Example 7.2.2 flowchart for primes

This flowchart and code show an algorithm to determine which numbers between 1 and 48 are prime numbers.

We already know that 2, 3, and 5 are prime, so the for loop starts at k = 6.

For each value it tests, it displays a message stating whether or not it is a prime number. It stores the prime numbers it finds in a vector called prime_list.


for_loop_primes_flowchart
"for loop" example with "if-then-elseif" logic:
For numbers < 7² = 49, we only need to test for divisibility by 2, 3, and 5
Initialize prime_list = [2, 3, 5]
K = 6
Display Messages:
if
(k is divisible by 2)
True
k is not prime.
It is divisible by 2
False
elseif
(k is divisible by 3)
True
k is not prime.
It is divisible by 3
False
elseif
(k is divisible by 5)
True
k is not prime.
It is divisible by 5
False
k is prime
Append k to prime_list
k=k+1

Try coding this logic.

Note that the for-loop automatically does the "k = k+1" shown at the bottom of the flowchart; do not code that line.

(There are more sophisticated and efficient prime-number algorithms; this one is an illustration of moderately complex logic.)

Solution

```
%% "for loop" example with “if-then-elseif” logic:
% Find the prime numbers between 1 to 48:
prime_list = [2,3,5] % Initial set of primes
for k = 6:48
    if( mod(k,2) == 0) % Tests if k is divisible by 2
        disp([num2str(k),' is not prime. It is divisible by 2'])
    elseif (mod(k,3) == 0) % Tests if k is divisible by 3
        disp([num2str(k),' is not prime. It is divisible by 3'])
    elseif( mod(k,5) == 0) % Tests if k is divisible by 5
        disp([num2str(k),' is not prime. It is divisible by 5'])
    else
        disp([num2str(k),' is prime.'])
        % Add k to the list of primes:
        prime_list = [prime_list, k];
    end % end if
end % for k = 6:48
% Display the primes found
prime_list
```

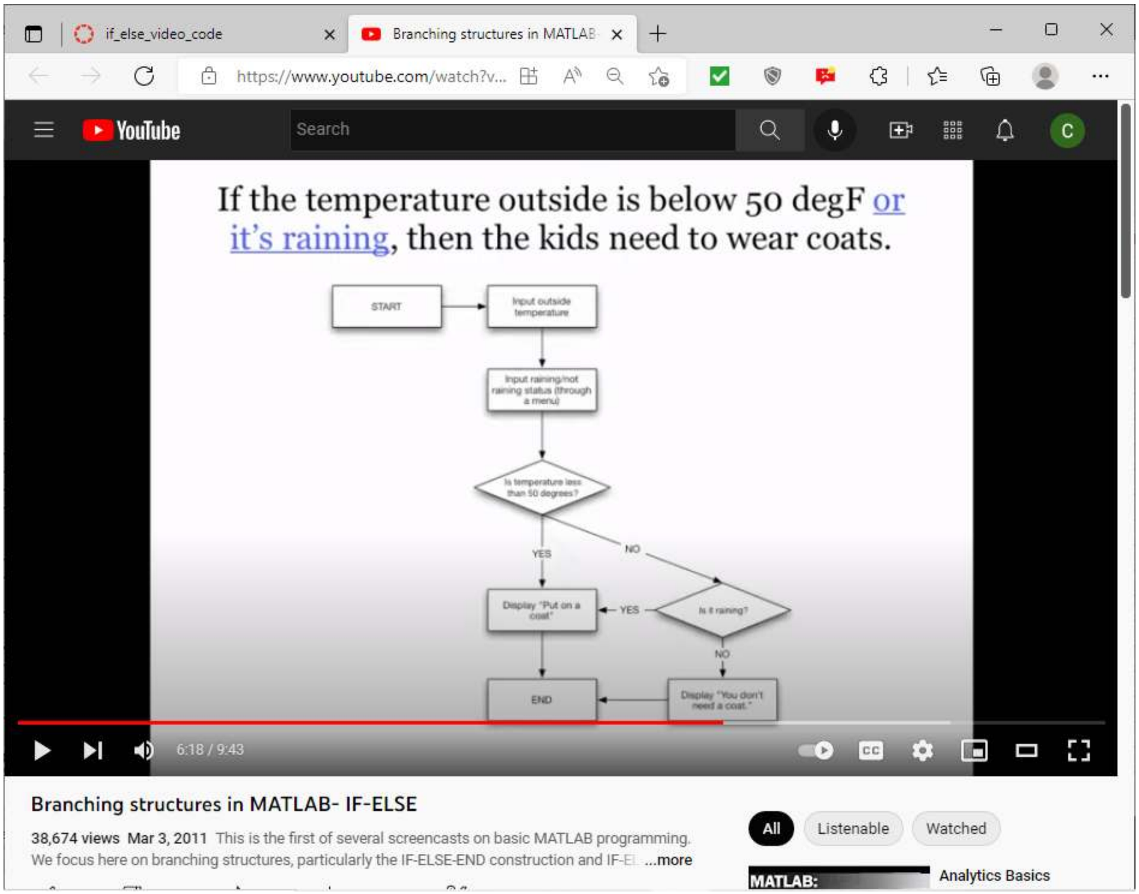
This code is the attached file for_loop_primes.m

.

7.2.1: Homework:

1. Watch the following video from Robert Talbert

Then code the second example in the video about the temperature and rain. The corresponding flowchart is this:


if_else_video_code
Branching structures in MATLAB
https://www.youtube.com/watch?v...
YouTube
Search
C
START
Input outside temperature
Input raining/not raining status (through a menu)
Is temperature less than 50 degrees?
YES
NO
Display "Put on a coat"
YES
Is it raining?
NO
END
Display "You don't need a coat."
6:18 / 9:43
CC
All
Listenable
Watched
MATLAB:
Analytics Basics

If the temperature outside is below 50 degF or it's raining, then the kids need to wear coats.

Branching structures in MATLAB- IF-ELSE

38,674 views Mar 3, 2011 This is the first of several screencasts on basic MATLAB programming. We focus here on branching structures, particularly the IF-ELSE-END construction and IF-E ...more

Figure \(\PageIndex{i}\): Flowchart for Temperature, Rain, Wearing a Coat.

Put parentheses around the conditional expressions, even though the video did not do this. Examples:

if (temp < 50)

elseif (raining == 1)

.

This page titled 7.2: Flowcharts is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

7.3: The input() Function

The input() function lets the use enter data from the command window (aka console)
It displays a message and stores the user's input

✓ Example 7.3.1 Input a number:

```
hours_sleep = input('How many hours did you sleep last night? ')
```

The message is a character string. It is enclosed in single quotes.

This line of code expects a number to be entered.

The space after the ? puts a space between the message and the user's answer.

Solution

Add example text here.

✓ Example 7.3.2 Input a character string

The default input is a number, so we need a 2nd parameter, 's', to tell input() to store a character string.

```
name2 = input('Enter the last of someone famous: ','s')
```

This line of code expects a character string to be entered.

Again, the space after the ? puts a space between the message and the user's answer.

Solution

Add example text here.

The next example is from Troy Siemers, Section 6.1.

✓ Example 7.3.3 Input name and age

Here is a simple section of code to collect a person’s first name and their age.

```
>> FirstName = input('Enter your first name : ','s');
>> Age = input('Enter your age : ');
```

Once executed, the user enters values for each, say

Enter your first name : Bob
Enter your age : 25

In this example, the 's' in the first input indicates that the program expects text input.

The second input (without the 's') indicates that the program expects numeric input.

When executed, the lines collect input one at a time. The space after the semicolon in each input() function statement separates what the user enters from the input() message.

When this code is run, each input() message is displayed and waits for the user to ente responses--in this case the name, Bob, and age, 25. Due to the semicolons, there is no output to the screen, but the values have been stored in the variables FirstName and Age (see the Workspace window to confirm).

Solution

Add example text here.

This part is by Carey A. Smith

Review section 4.7 on why and how numbers are converted to characters strings in order to combine them in a message. It also explains concatenation.

? Exercise 7.3.1 Input and Concatenation

Put the following code into a script.

Run the code and enter answers to the input() questions.

```
first = input('Enter your first name: ','s')
last = input('Enter your last name: ','s')
year = input('Enter your year in college as a number: ')
disp([first,' ',last,', college year= ',num2str(year)]) % The ' ' inserts a space
between first and last.
```

% num2str(year) converts the number variable, year, to a character string. See Section 4.7.

Answer

Add texts here. Do not delete this text first.

7.3.1: Homework:

? Exercise 7.3.2 Day of the week

Create a script file for the following:

[2 pts] Use the input() function to ask the user to enter the day of the week as a character string. The user's response is stored in a variable named day. Remember, that when the expected input value is a character string (not a number), you need to use the 's' optional argument, such as this:

```
day = input('Enter the day of the week: ', 's')
```

[2 pts] Use concatenation with the disp() function to display on the screen a message similar to this:

'Today is Monday'. (Use the variable day.)

Answer

Add texts here. Do not delete this text first.

.

? Exercise 7.3.3 Temperature

Create a script file for the following:

[2 pts] Use the input() function to ask the user to enter the temperature as a number, not a string. Don't use the 's' option for this. The user's response is stored in a variable named T.

[2 pts] Use concatenation with the disp() function to display on the screen a message similar to this:

'The temperature = 68 degrees'. Use num2str(T) to display the temperature.

Answer

Add texts here. Do not delete this text first.

.

This page titled 7.3: The input() Function is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

7.4: Compound Logical Operators

To create compound statments of the relational operators, we can combine these using the logical operators. The truth or falsity of these follows basic rules of logic, so it helps to have some knowledge of truth tables. Again, read them as questions!

`&` (“and” returns true if both parts are true)
`~` (“not” returns true if the initial value is false)
`|` (‘or” returns true if either part is true)
`xor` (“exclusive or” returns true if either part is true, but NOT both true)

Example 7.2.1: Basic Logic

```
>> (3 >= 2) & (3+4 == 6)

ans=
0
```

Here, the question “is 3 greater than or equal to 2 AND 3 plus 4 equal to 6?” is answered as false (or a zero) since even though 3 is greater than 2, it is not true that 3 plus 4 is 6.

Note that there are also the operators && and || which also mean “and” and “or” but are called “short-circuited” operators (look it up). As you work with these in the Editor, orange lines on the right side of the screen may appear suggesting you use && in place of & (or vice-versa) and || in place of | (or vice versa). Just take those suggestions and you’ll be fine. These operations extend to matrices with an entry-by-entry comparison of matrices of the same size.

Example 7.2.2

```
>> a=[0,1;1,2], b=[0,0;1,1]

a =
0 1
1 2
b =
0 0
1 1
```

with

```
>> a&b, a|b, xor(a,b)
```

returns

```
ans =
0 0
1 1

ans =
0 1
1 1

ans =
0 1
0 0
```

One minor point to note is that while Matlab treats 0 as false, any other positive whole number is considered as true. So, in this example, even though the (2, 2) entry of a is a 2, it is considered as “true” for logical comparison.

This page titled 7.4: Compound Logical Operators is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Troy Siemers (APEX Calculus) via source content that was edited to the style and standards of the LibreTexts platform.

- **7.3: Logical Operators** by Troy Siemers is licensed CC BY-NC 3.0. Original source: https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

7.4.1: Compound if, elseif Logic Examples

When we want to test if a variable is between 2 values, we can use compound if or elseif logic.

Example 7.4.1.1 BodyTemperature Compound if-else with 'or' logic

```
% Body Temperature version 1
% Normal body temperatures are between 36.1°C and 37.2°C
% https://medlineplus.gov/ency/article/001982.htm
% For this script, the user will input their body temperature:
body_temp = input('Enter your body temperature in degrees C: ')
if ((body_temp < 36.1) | (body_temp > 37.2))  % if (body_temp < 36.1) or if  (body_temp > 37.2)
  disp('Your body temperature is outside the normal range.')
  if (body_temp < 36.1)
    disp('It is lower than normal.')
  else
    disp('It is higher than normal.')
    disp('You may have an infection or illness.')
  end
else
    disp('Your body temperature is normal.')
end
```

Solution

Add example text here.

.

Example 7.4.1.2 Divergent Car Control

This is an example of a divergent series. Instead of the terms getting smaller, these terms get larger in absolute value. The loop is broken when the terms get too large.

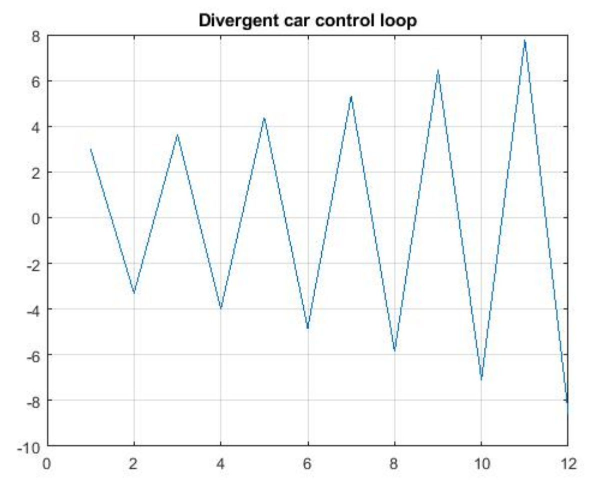
% Divergent loop due to unstable feedback
This models vehicles on a freeway with 3 lanes.
Initial conditions:
Car 1 is in freeway lane 1, on the left next to the center divider.
A semi truck is in lane 2
Car 3 is in freeway lane 3, on the right next to the shoulder.
Cars 1 and 3 can't see each other becuz of the semi truck.
Action:
Car 1 passes the truck and begins to move to the right into lane 2.
Car 3 also passes the truck begins to move to the left into lane 2
So the driver of car 1 turns hard to the left.
Due to the speed of the car and the delay to realize how far he has turned,
each time he "over corrects"--that is, he turns too far each time. This is called unstable feedback.

x is the number of feet car 1 he has moved to the right, measured from the center of lane 3
Negative values of x are to the left of the center of lane 3.

The for loop is broken if either the change in position, dx, gets too large or if the total position error gets too large.

Solution

```
% Initialize
x(1) = 3; % Position at time 1 when he make his 1st correction
r = 2.1;  % Over correction factor
for m = 2:20  % m is the interation index
% The change in position each iteration is
   dx = -r*x(m-1);
   % Add this to the current position:
   x(m) = x(m-1) + dx;
   % If either the absolute value of dx is greater than 15,
   % or the absolute value of x is greater than 10
   %  then use the "break" keyword to stop looping.
   if ((abs(dx) > 15) | (abs(x(m)) > 10))
      break;
   end
end
```



.

.

This page titled 7.4.1: Compound if, elseif Logic Examples is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

7.5: Working with Conditions

Some of the loops in the previous chapter don't work if the value of n isn't set correctly before the loop runs. For example, this loop computes the sum of the first n elements of a geometric sequence:

```
A1 = 1;
total = 0;
for i=1:n
    a = A1 * 0.5^(i-1);
    total = total + a;
end
ans = total
```

It works for any positive value of n, but what if n is negative? In that case, you get:

```
total = 0
```

Why? Because the expression 1:-1 means "all the numbers from 1 to -1, counting up by 1." It's not immediately obvious what that should mean, but MATLAB's interpretation is that there aren't any numbers that fit that description, so the result is

```
>> 1:-1
ans = 1x0 empty double row vector
```

The result is an empty vector with one row and 0 columns. If you loop over an empty vector, the loop never runs at all, which is why in this example the value of total is zero for any negative value of n.

If you're sure that you'll never make a mistake, and that the preconditions of your functions will always be satisfied, then you don't have to check. But for the rest of us, it's dangerous to write a script, like this one, that quietly produces the wrong answer (or at least a meaningless answer) if the input value is negative. A better alternative is to use an if statement.

if Statements

The if statement allows you to check for certain conditions and execute statements if the conditions are met. In the previous example, we could write:

```
if n<0
    ans = NaN
end
```

The syntax is similar to a for loop. The first line specifies the condition we're interested in; in this case we're asking if n is negative. If it is, MATLAB executes the body of the statement, which is the indented sequence of statements between the if and the end.

MATLAB doesn't require you to indent the body of an if statement, but it makes your code more readable, so you should do it. Don't make me tell you again.

In this example, the "right" thing to do if n is negative is to set ans = NaN, which is a standard way to indicate that the result is undefined (not a number).

If the condition is not satisfied, the statements in the body are not executed. Sometimes there are alternative statements to execute when the condition is false. In that case you can extend the if statement with an else clause.

The complete version of the previous example might look like this:

```
if n<0
    ans = NaN
else
    A1 = 1;
    total = 0;
    for i=1:n
        a = A1 * 0.5^(i-1);
        total = total + a;
    end
    ans = total
end
```

Statements like if and for that contain other statements are called *compound* statements. All compound statements end with... end.

In this example, one of the statements in the else clause is a for loop. Putting one compound statement inside another is legal and common, and sometimes called *nesting*.

Relational Operators

The operators that compare values, like are called *relational operators* because they test the relationship between two values. The result of a relational operator is one of the *logical values*: either 1, which represents “true,” or 0, which represents “false.”

Relational operators often appear in if statements, but you can also evaluate them at the prompt:

```
>> x = 5;
>> x < 10
ans = 1
```

You can assign a logical value to a variable:

```
>> flag = x > 10
flag = 0
```

A variable that contains a logical value is often called a *flag* because it flags the status of some condition.

The other relational operators are =, which are self-explanatory, ==, for “equal,” and `~=`, for “not equal.”

Don’t forget that == is the operator that tests equality, and = is the assignment operator. If you try to use = in an if statement, you get an error:

```
>> if x=5
 if x=5
     |
Error: Incorrect use of '=' operator.
To assign a value to a variable, use '='.
To compare values for equality, use '=='.

Did you mean:
>> x = 5
```

In this case, the error message is pretty helpful.

Logical operators

To test if a number falls in an interval, you might be tempted to write something like 0 < x < 10, but that would be wrong, so very wrong. Unfortunately, in many cases, you'll get the right answer for the wrong reason. For example:

```
>> x = 5;
>> 0 < x < 10            % right for the wrong reason
ans = 1
```

But don't be fooled!

```
>> x = 17
>> 0 < x < 10            % just plain wrong
ans = 1
```

The problem is that MATLAB is evaluating the operators from left to right, so first it checks if 0<x. It is, so the result is 1. Then it compares the logical value 1 (not the value of x) to 10. Since 1<10, the result is true, even though x is not in the interval.

For beginning programmers, this is an evil, evil bug!

One way around this problem is to use a nested if statement to check the two conditions separately:

```
ans = 0
if x>0
    if x<10
        ans = 1
    end
end
```

But it's more concise to use the AND operator, &&, to combine the conditions.

```
>> x = 5;
>> x>0 && x<10
ans = 1

>> x = 17;
>> x>0 && x<10
ans = 0
```

The result of AND is true if *both* of the operands are true. The OR operator, ||, is true if *either or both* of the operands are true.

...

However, you have to be careful with the range of the loop. In the previous version, the loop runs from 3 to n, and each time we assign a value to the ith element.

It would also work to "shift" the index over by two, running the loop from 1 to n-2:

```
F(1) = 1
F(2) = 1
for i=1:n-2
    F(i+2) = F(i+1) + F(i)
end
```

Either version is fine, but you have to choose one approach and be consistent. If you combine elements of both, you'll get confused. I prefer the version in Listing [lst:fib_vec] that has F(i) on the left side of the assignment, so that each time through the loop it assigns the ith element.

Everything is a Matrix

In math (specifically in linear algebra) a vector is one-dimensional and a *matrix* is two-dimensional.

But in MATLAB, everything is a matrix (except strings). You can see this if you use the whos command to display the variables in the workspace. whos is similar to who, but it also displays the size of each value and other information.

To demonstrate, I'll make one of each kind of value:

```
>> number = 5
number = 5

>> vector = [1 2 3 4 5]
vector = 1     2     3     4     5

>> matrix = ones(2,3)
matrix =
     1     1     1
     1     1     1
```

The built-in function ones builds a new matrix with the given number of rows and columns, and sets all the elements to 1. Now let's see what we've got.

```
>> whos
  Name          Size                Bytes  Class     Attributes

  number        1x1                     8  double
  vector        1x5                    40  double
  matrix        2x3                    48  double
```

According to MATLAB, all three are matrices. They have the same class, double, which means they contain double-precision floating-point numbers.

But they have difference sizes: the size of number is 1x1, which means it has 1 row and 1 column; vector has 1 row and 5 columns; and matrix has 2 rows and 3 columns.

So, you can think of your values as numbers, vectors, and matrices, but in MATLAB they are all matrices.

Search

Yet another use of loops is to search the elements of a vector and return the index of the value you're looking for (or the first value that has a particular property).

For example, the loop in Listing [lst:vec_search] finds the index of the element 0 in X:

```
for i=1:length(X)
    if X(i) == 0
        ans = i
    end
end
```

A funny thing about this loop is that it keeps going after it finds what it’s looking for. That might be what you want; if the target value appears more than one, this loop provides the index of the *last* one.

But if you want the index of the first one (or you know that there is only one), you can save some unnecessary looping by using the break statement.

```
for i=1:length(X)
    if X(i) == 0
        ans = i
        break
    end
end
```

break does pretty much what it sounds like. It ends the loop and proceeds immediately to the next statement after the loop (in this case, there isn’t one, so the code ends).

This page titled 7.5: Working with Conditions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **4.10: Working with Conditions** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

7.6: Shortcut and, or, Logical Operations

Consider the following compound logical code:

```
if ( (n >= 4) & (abs(n) <= 10) )
```

This will be true for values of n between 4 and 10 inclusive.

You may see code similar to this version:

```
if ( (n >= 4) && (abs(n) <= 10) )
```

The "&&" means that if (n >= 4) is false, then MATLAB will skip the check of (abs(n) <= 10), because the compound expression will be false, whether (abs(n) <= 10) is true or false. In this example, it is not necessary to use &&; the & works just fine. "&&" is only needed in a case like this:

```
if ( (n ~= 0) && (log10(n) <= 0.5) )
```

because we would want to avoid computing log10(0).

It is similar for | vs. ||

```
if ( (n < -3) || (x > 2) )
```

If the 1st logical expression is true, then there is no need to test the 2nd logical expression, because the compound "or" expression is true whenever either part is true. For "or" compound statements, the double "||" is not necessary; you can use the single "|" in all cases.

Tip: for readability, put parentheses around each logical expression, as is shown here. It runs just as fast and avoids both incorrect operations and misunderstandings of the code. Readable code is debuggable code.

.

This page titled 7.6: Shortcut and, or, Logical Operations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

7.7: for loops with break Logic

By Carey A. Smith

7.7.1: Converging Series with if and break Logic

A series is the sum of the terms of a sequences. They are typically computed with for loops.

(When the terms of the sequence have already been computed, then the series can be computed with the sum() function. This is an example of how MATLAB can simplify a calculation.)

A series is convergent (or converges) if the sequence of its partial sums tends to a limit. (https://en.Wikipedia.org/wiki/Convergent_series) Maclaurin's series generally converge for values of x near zero. (A Maclaurin series is a Taylor series derived for points near 0.)

This subsection describes for loops that are terminated when the terms become very small. Not all series whose terms become small are convergent.

The MATLAB keyword "break" terminates the execution of a for loop or a while loop. In nested loops, break exits only from the loop in which it occurs. Control passes to the statement that follows the end of that loop.

Example 7.7.1 Power Series

Consider the summation of this power series

$$\sum_{n=0}^{\infty} a_n x^n = a_0 + a_1 x + a_2 x^2 + \ldots.$$

where an = (3/(n+1))

Since sum() is an important built-in function, we won't use "sum" as a variable. We will use "total" instead.

Although this series has an infinite number of terms, the terms will get very small as n gets large.

We will end the summation when the absolute value of nth term is < 0.0001

.

Solution

```
% This is a script to sum a power series
x     = 0.3; % This is a precondition
total = 0;   % Initialize total before the loop.
term_limit = 0.0001;
for n = 1:1000
    term = (3/(n+1))*x^n;
   total = total + term;
   if(abs(term) < term_limit)
      break;
   end
end
disp(['Loop broke at n = ',num2str(n)])
disp(['Power Series total = ',num2str(total)])
```

This code displayed these messages:

Loop broke at n = 7
Power Series total = 0.56672

.

✓ Example 7.7.2 Cosine Taylor's series

The cosine Maclaurin series starts with these terms:

$$cosx = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} \dots \tag{7.7.1}$$

The index of the for loop is m.

This terms in this series have alternating signs, which we get by multiplying by (-1)^m.

(-1)^m = -1 when m is odd, and (-1)^m = 1 when m is even.

This series has only even powers of x, so we will use k = 2* m for the powers.

The $m^(th)$ term of the cosine Taylor's series is: $(-1)^m * \frac{x^k}{k!}$

The break keyword will terminate the sum when abs(terms(m)) < 0.001

Here's the function:

```
function cos1 = my_cos(x)
% Taylor's series approximation

terms = zeros(1,100); % preallocate

% Usually a the total for a summation is initialize to 0.

% For the cosine series, the term for m = 0 is 1, becuz x^0 = 1, factorial(0) = 1.

So we initialize cos1 (the total) = 1.
cos1 = 1; % Initialize sum of terms in the Taylor's series
terms(1) = cos1;
for m = 1:100 % n = m-1
    k = 2*m; % Only even powers of x are used
    terms(m) = (-1)^m*x^k/factorial(k);
    % Note that (-1)^m = -1 when m is odd
    % and it = 1 when m is even.
    cos1 = cos1 + terms(m);
    if(abs(terms(m)) < 0.001) % Terminate the series when terms(m) is small
        break;
    end
end
```

Solution

Add example text here.

.

? Exercise 7.7.1 Convergent Series Homework 1

Write a script that uses a for loop to sum the terms of the series whose mth term = $\frac{1}{m^2}$

Use "total" as the variable name. Do not use "sum" as a variable name, because that is an important built-in function.

Initialize total = 0 before the for loop.

Write a for loop that starts with:

for m = 1:1000

Break the for loop when the absolute value of term is < 0.0002

Answer

Add texts here. Do not delete this text first.

.

7.7.2: Diverging Series with if and break Logic

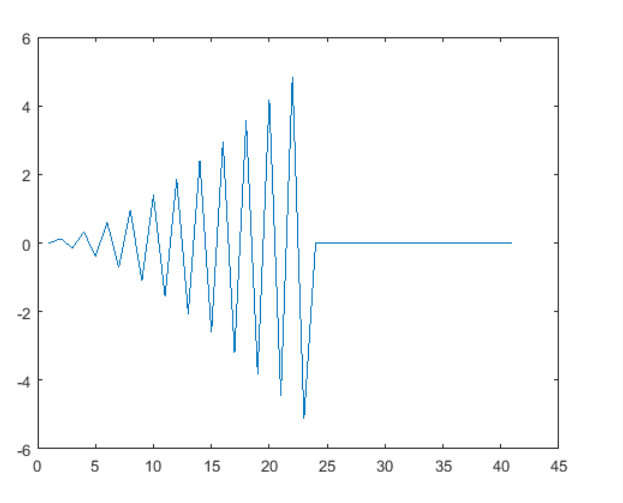
This subsection describes for loops that are terminated when the terms become very large.

Example 7.7.3 Divergent for loop

```
clear all; close all; format compact; clc;
n = 40; % maximum number of iterations
total = zeros(1,(n+1));
term_lim = 10; % Terminate the sequence if termk > term_lim
for k = 2:n
    termk = (-1)^k*(k^2)/(k+30);
    if( abs(termk) > term_lim)
        disp(['k= ',num2str(k),', termk = ',num2str(termk),...
            '>10, so break the loop'])
        break;
    else
        total(k) = total(k-1) + termk;
    end
end

figure;
plot(total)
```

Solution

The plot should look like this figure:



```
Figure \(\PageIndex{i}\): Divergent for loop
```

A series can diverge, even though the terms become small. The harmonic series is an example of a series which diverges because the terms do not become small fast enough. This is an example

$$harmonic = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \dots \tag{7.7.2}$$

7.7.3: Homework Exercises

TBD

.

This page titled 7.7: for loops with break Logic is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

7.8: 'while' Loops

We review `for` loops and introduce `while` loops in this section.

Suppose we want to have part of our program re-run a preset number of times. For this, we use a `for` loop. The basic format of this structure is:

```
for (put counter conditions here)
(put calculations here)
end
```

✓ Example 7.4.1: Comparing 'for' and 'sum'

Let’s compare two methods for adding up the first five integers. Using the sum command we can use

```
>> sum(1:5)
ans =
15
```

Now, using a for loop to create a cumulative sum:

```
totalsum=0; % initialize
for i=1:5
    totalsum=totalsum+i;
end
disp(totalsum)
```

The variable `totalsum` will have value 15.

In example 7.4.1, the `for` loop was the long way of doing the problem (and therefore stresses the power of the sum function), but the following example shows a more in-depth `for` loop.

✓ Example 7.4.2: Using 'for'

Let’s find the first 10 Fibonacci numbers using the recursive definition.

```
%initialize the matrix
A=zeros(1,10);
A(1)=0;
A(2)=1;

for i=3:10
    A(i)=A(i-1)+A(i-2);
end

disp(A)
```

The output for this would be

```
A =
0 1 1 2 3 5 8 13 21 34
```

Now, suppose want to have part of our program run until a certain condition is met, even though we may not know how many times the loop will need to run until that happens. For this, we use a while loop.

The basic format of this structure is:

```
while (put conditions for the loop to keep running)
    (put calculations here)
```

```
end
```

Let’s rewrite the last example with a slight twist. Let’s find the Fibonacci numbers until they exceed 1000.

✓ Example 7.4.3: Using 'while'

```
%Initialize the matrix. (We don't know how big it will be,
% so we will grow it in the while loop.)
A(1)=0;
A(2)=1;
j=2; %initialize a counter
while A(j) < 1000
    j=j+1; %move the counter along
    A(j)=A(j-1)+A(j-2);
end
```

```
disp(A)
```

The output for this would be

```
A =
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The loop concludes when we pass 1000.

.

When using a while loop, avoid coding an infinite loop. If an infinite loop occurs, you can stop the program by pressing Ctrl-C.

This silent video is helpful to understand the use of break, continue, and return in while and for loops.

https://www.youtube.com/watch?v=lpBFyxolJS8

Break and Continue Statements in Matlab

.

This page titled 7.8: 'while' Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Troy Siemers (APEX Calculus) via source content that was edited to the style and standards of the LibreTexts platform.

- **7.4: 'for' and 'while' Loops** by Troy Siemers is licensed CC BY-NC 3.0. Original source: https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

7.9: 'switch' Logic

This part is by Troy Siemers.

We review the `if\elseif\else` construct and introduce the `switch\case\otherwise` construct in this section.

Suppose we want to have part of our program run only under certain conditions. For this, we use an if\else structure. The basic format of this structure is:

```
if (put condition(s) here)(put calculations here to be run if the conditions are met)
elseif (other condition(s))(calculations that will run under the new conditions)(more
elseif statements, if desired)
else (calculations run if none of the previous conditions are met)
end
```

✓ Example 1: Using `if`

Let's write a script M-file that lets a user input a number and then displays if that number is less than 5, between 5 and 10 (inclusive), or greater than 10.

```
number = input('Input a number ');
if (number < 5)
disp('Your number is less than 5.')
elseif ((number >=5) && (number <= 10))
disp('Your number is between 5 and 10.')
else
disp('Your number is greater than 10.')
end
```

Try running this program using various numbers for input.

The switch\case structure is similar to the if structure, but has a few advantages. First of all it is easier to read and second, it is better if you are comparing strings (of possibly different lengths). The basic format is:

```
switch (expression to test)
case (case condition)
(output in that case)
case (case condition)
(output in that case)

...(more cases)
otherwise
(do this if no cases are met)

end
```

✓ Example 2: Using `switch`

Let's check to see if a cadet is in his\her first two years at VMI.

```
Year = 'second class';

switch Year
case {'fourth class','third class'}
disp('You are in the first two years.')
case {'second class','first class'}
disp('You are in the last two years.')
```

```
otherwise
disp('You must be a 5th year.')
end
```

The cases are grouped by curly brackets so that a case will be satisfied if the value of Year is any of the values in a specific case. Once this code is executed, the switch command will look at the value of Year and the output should be

```
You are in the last two years.
```

.

This part is by Carey Smith

Example 7.9.3 Switch_Major.m

This script uses the menu() function which displays a list of choices. The user clicks on one of the menu items. This converts the selection to a number stored in the variable Major.

The switch structure uses the Major variable value to determine a message to display.

```
% Switch_Major.m
Major = menu('Select a Major: ', ...
'Physics', 'Engineering', 'Math', 'Chemistry', ...
'History','Psychology', ...
'English', 'Art', 'Buisiness')
switch Major
case {1,2,3,4}
disp('That is a STEM major')
case {5,6}
disp('That is a Social Science major')
otherwise
disp('That is neither a STEM nor a Social Science major')
end
```

Solution

Add example text here.

.

Exercise 7.9.1 Switch Shop Hours

Review the Switch_Major.m code above, then due this assignment.

Create a switch script named Switch_Shop_Hours_YourName.m

The user will chose a day of the week, and the script will display the hours a shop is open
or a message that the shop is closed.

Use the following menu function to allow the user to select a day of the week:

```
day = menu('Select a day of the week: ',...
'Sun', 'Mon', 'Tues','Weds', ...
'Thurs', 'Fri', 'Sat')
```

Then write switch logic using "day" as the switch variable.
[Day will be an integer from 1 to 7]
For the cases that the day is 1 or 2 (Sun or Mon), display:
'The shop is closed'

For the cases that the day is 3, 4, or 5 (Tues, Weds, or Thurs), display:
'The shop is open 8am to 5:30pm'
For the case that the day is 6 (Fri), display:
'The shop is open 8am to 7:00pm'
For the case that the day is 7 (Sat), display:
'The shop is open 8am to noon'

Answer

Add texts here. Do not delete this text first.

.

This is a video on the switch statement:

https://www.youtube.com/watch?v=zqGJE2WGHq4

Switch Statement in Matlab, Learnrope Tutorials

.

This page titled 7.9: 'switch' Logic is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith via source content that was edited to the style and standards of the LibreTexts platform.

SECTION OVERVIEW

7.10: The find() function

The MATLAB built-in function find() is an efficient method to find the indices of data that satisfy some logical condition.

The syntax of the find function is:

```
indx = find( condition )
```

The output of the find() function is the set of indices for which the "condition" is true.

This is illustrated by these examples:

Example 7.10.1 find() with an == condition

```
a = 1:0.5:4 % a = [1.0 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]
idx1 = find(a == 3.0) % Find the elements equal to 3 and return their indices
% idx1 = 5, becuz the 5th element of a is 3.0

% Check it with this code
a_idx1 = a(idx1) % = 3.0
```

Solution

Add example text here.

.

Example 7.10.2 find() with a < condition

```
a = 1:0.5:4 % Same vector as above
idx2 = find(a < 3.0)
% idx = 1 2 3 4
% These are the indices of the elements of a < 3

% Check it with this code
a(idx2) % = 1.0, 1.5, 2.0, 2.5
```

Solution

Add example text here.

.

Example 7.10.3 find() with a compound condition

```
a = 1:0.5:4 % Same vector as above
% Find the indices of the elements between 2.0 and 3.5, inclusive
% We need to use a compound conditional expression
idx3 = find( (2.0 <= a) & (a <= 3.5) )
% "&" = "and"
% That means that for a particular value of a, both conditional parts have to be true for
% for the compound conditional to be true.
% idx3 = 3 4 5 6
% These are the indices of the elements that satisfy both parts
% Check it with this code
```

```
a(idx3) % = 2.0, 2.5, 3.0, 3.5
```

Solution

Add example text here.

.

Example 7.10.4 The result of the compound condition itself

```
%% The condition by itself:
( (2.0 <= a) & (a <= 3.5) )
%ans =
% 1×7 logical array
% 0 0 1 1 1 1 0
% 0= False for the elements that do not satisfy both parts of the
% conditional
% 1=True for the elements that do satisfy both parts
```

Solution

Add example text here.

.

Example 7.10.5 What could go wrong?

```
%% What happens if we try this?
% A single condition, instead of a compound condition:
idx3b = find(2.0 <= a <= 3.5)
```

```
% The result is: idx3b = 1 2 3 4 5 6 7 (!?)
% This is clearly not what was desired.
```

```
% The condition by itself (2.0 <= a <= 3.5) returns:
% 1 1 1 1 1 1 1
% becuz every value of a is either >= 2 or <= 3.5
% You can only have a comparison between 1 number and 1 variable
% or 2 variables.
% You need to have each part of the compound be a complete condition by itself, as above, namely:
```

```
% ( (2.0 <= a) & (a <= 3.5) )
```

Solution

Add example text here.

.

Example 7.10.6 find() with 2 variables

```
a = 1: 0.5 :4 % As above
b = 2
idx4 = find(b == a)
% idx4 = 3
a(idx4) % 2
```

Solution

Add example text here.

.

✓ Example 7.10.7 find() with no match.

When there are no elements that satisfy the condition,

then the find() command returns an empty vector.

a = 1:0.5:4
b = 20
idx4 = find(b == a)

% idx4 = empty

Solution

Add example text here.

.

✓ Example 7.10.8

%% Example 5: 2 vectors of the same length
a = 1:0.5:4
% a = [1.0 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]
c = [0 1 2 3 4 5 6]
idx5 = find(a == c)
% idx5 = 3, because a(3) = b(3) = 2
% Also, even though both vectors have values of 1, 3, and 4
% these don't occur in the same place, so they are not matches.

Solution

Add example text here.

.

? Exercise 7.10.1 Test Scores >= 90

% Use these test scores:
test_scores = [
73
72
88
94
77
92
88
65
86
78
97
65]
% Use the find() function to find and list the test scores >= 90

Answer

Add texts here. Do not delete this text first.

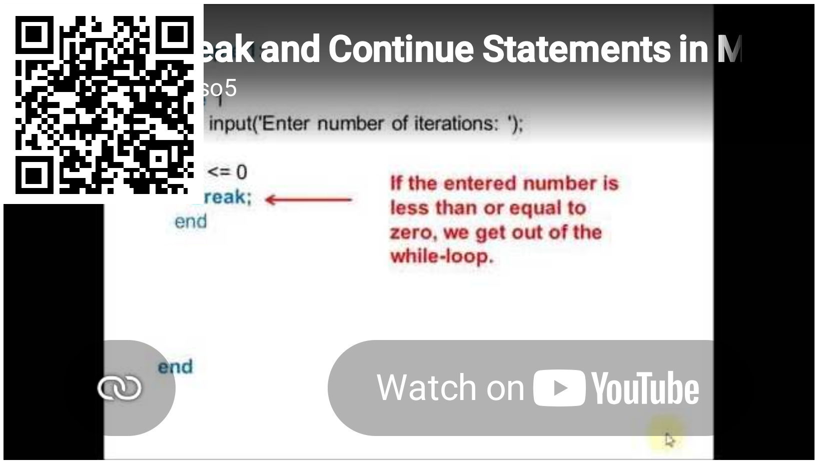
.

This page titled 7.10: The find() function is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

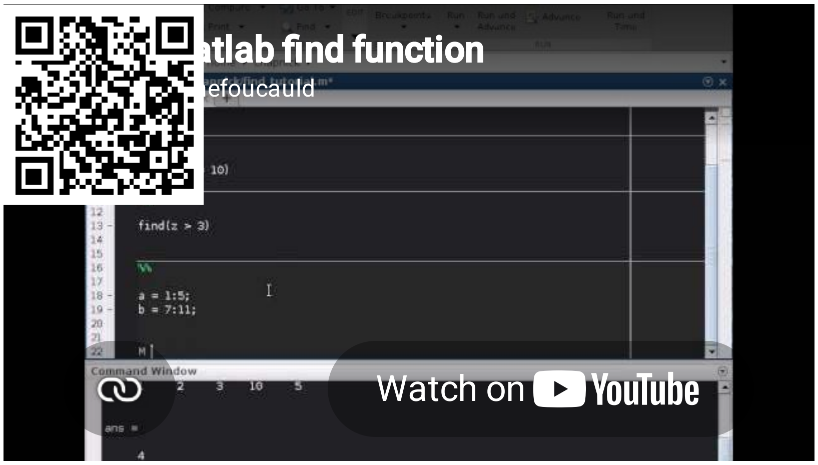
7.11: Chapter 7 Conditionals, Part 2, Videos English and Espanol

English

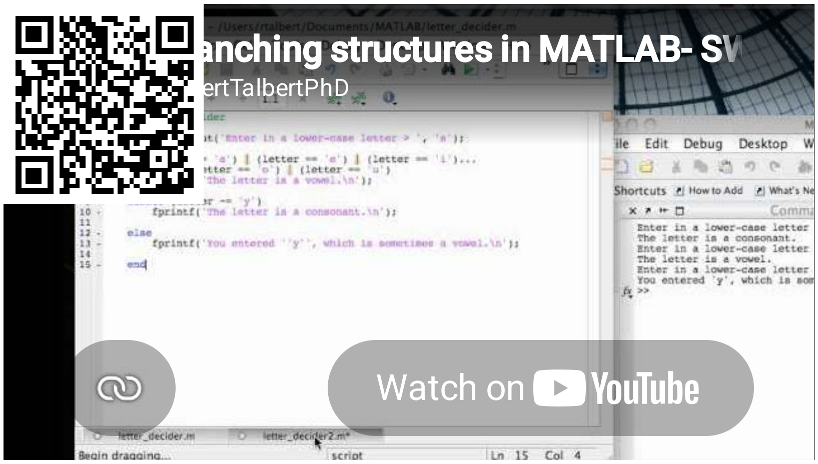
Break and Continue Statements in Matlab


eak and Continue Statements in M
so5
input('Enter number of iterations: ');
<= 0
reak;
end
If the entered number is less than or equal to zero, we get out of the while-loop.
end
Watch on YouTube

Matlab find() function video


atlab find function
efoucauld
find(z > 3)
a = 1:5;
b = 7:11;
Command Window
ans =
Watch on YouTube

SWITCH-CASE video RTalbert

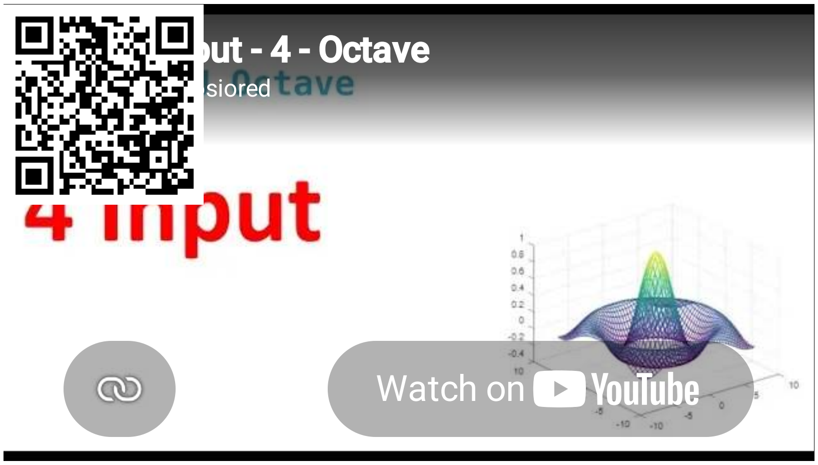

anching structures in MATLAB- SV
ertTalbertPhD
File Edit Debug Desktop
Shortcuts How to Add What's New
letter_decider.m
letter_decider2.m*
Begin dragging...
script
Ln 15 Col 4
Watch on YouTube

Switch Statement in Matlab (learnrope.com)

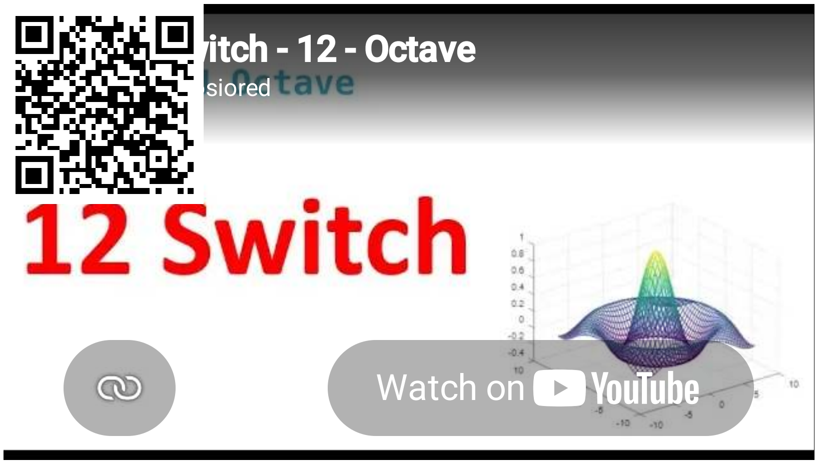
itch Statement in Matlab
nrope Tutorials
Switch Statement
Matlab
L Rope
Watch on YouTube

Espanol

Input - 4 - Octave


put - 4 - Octave
siored
4 Input
Watch on YouTube

Switch - 12 - Octave


itch - 12 - Octave
siored
12 Switch
Watch on YouTube

Ciclo while - 24 - Octave

clo while - 24 - Octave
siored
tave
Ciclo while
Watch on YouTube

This page titled 7.11: Chapter 7 Conditionals, Part 2, Videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

8: User-defined Functions of Vectors

Now that we have functions and vectors, we’ll put them together to write functions that take vectors as input variables and return vectors as output variables. You’ll also see two patterns for computing with vectors: existential and universal quantification.

- 8.1: Functions and Vectors
  - 8.1.1: Built-in MATLAB Functions of Vectors
  - 8.1.2: User-Defined Functions of Vectors Examples and Exercises
- 8.2: Computing with Vectors
- 8.3: Debugging in Four Acts
  - 8.3.1: Debugging Theorems all in 1 Place
- 8.4: Chapter Review
- 8.5: Chapter 8 Function Videos English and Espanol

This page titled 8: User-defined Functions of Vectors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith via source content that was edited to the style and standards of the LibreTexts platform.

8.1: Functions and Vectors

In this section we'll look at common patterns involving functions and vectors, and you will learn how to write a single function that can work with *vectors* as well as scalars.

Vectors as Input Variables

Since many of the built-in functions take vectors as arguments, it should come as no surprise that you can write functions that take vectors as input. Here's a simple (but not very useful) example:

```
function res = display_vector(X)
    for i=1:length(X)
        display(X(i))
    end
end
```

There's nothing special about this function. The only difference from the scalar functions we've seen is that this one uses a capital letter to remind us that `X` is a vector.

Using `display_vector` doesn't actually return a value; it just displays the elements of the vector it gets as an input variable:

```
>> display_vector(1:3)
    1
    2
    3
```

Here's a more interesting example that encapsulates the code from Listing 4.2 on page 4.7 to add up the elements of a vector:

```
function res = mysum(X)
    total = 0;
    for i=1:length(X)
        total = total + X(i);
    end
    res = total;
end
```

I called this function `mysum` to avoid a collision with the built-in function `sum` , which does pretty much the same thing.

Here's how you call it from the Command Window:

```
>> total = mysum(1:3)
total = 6
```

Because this function has an output variable, I made a point of assigning it to a variable.

Vectors as Output Variables

There's also nothing wrong with assigning a vector to an output variable. Here's an example that encapsulates the code from Listing 4.3 on page 4.7:

```
function res = mysquare(X)
    for i=1:length(X)
        Y(i) = X(i)^2;
    end
    res = Y;
end
```

This function squares each element of `X` and stores it as an element of `Y`. Then it assigns `Y` to the output variable, `res`. Here's how we use this function:

```
>> V = mysquare(1:3)
V = 1     4     9
```

The input variable is a vector with the elements `1,2,3`. The output variable is a vector with the elements `1,4,9`.

Vectorizing Functions

Functions that work on vectors will almost always work on scalars as well, because MATLAB considers a scalar to be a vector with length 1.

```
>> mysum(17)
ans = 17

>> mysquare(9)
ans = 81
```

Unfortunately, the converse isn't always true. If you write a function with scalar inputs in mind, it might not work on vectors.

But it might! If the operators and functions you use in the body of your function work on vectors, then your function will probably work on vectors. For example, here's the very first function we wrote:

```
function res = myfunc(x)
    s = sin(x);
    c = cos(x);
    res = abs(s) + abs(c);
end
```

And lo! It turns out to work on vectors:

```
>> Y = myfunc(1:3)
Y = 1.3818    1.3254    1.1311
```

Some of the other functions we wrote don't work on vectors, but they can be patched up with just a little effort. For example, here's `hypotenuse` from Section 5.1:

```
function res = hypotenuse(a, b)
    res = sqrt(a^2 + b^2);
end
```

This doesn't work on vectors because the `^` operator tries to do matrix exponentiation, which only works on square matrices.

```
>> hypotenuse(1:3, 1:3)
Error using  ^  (line 51)
Incorrect dimensions for raising a matrix to a power.
Check that the matrix is square and the power is a scalar.
To perform element-wise matrix powers, use '.^'.
```

But if you replace `^` with the element-wise operator (`.^`), it works!

```
>> A = [3,5,8];
>> B = [4,12,15];
>> C = hypotenuse(A, B)

C = 5     13     17
```

The function matches up corresponding elements from the two input vectors, so the elements of `C` are the hypotenuses of the pairs $(3,4)$, $(5,12)$, and $(8,15)$, respectively.

In general, if you write a function using only element-wise operators and functions that work on vectors, the new function will also work on vectors.

Sums and Differences

Another common vector operation is *cumulative sum*, which takes a vector as an input and computes a new vector that contains all of the partial sums of the original. In math notation, if V is the original vector, the elements of the cumulative sum, C, are

$$C_i = \sum_{j=1}^{i} V_j$$

In other words, the ith element of C is the sum of the first i elements from V. MATLAB provides a function named `cumsum` that computes cumulative sums:

```
>> X = 1:5

X = 1      2      3      4      5

>> C = cumsum(X)

C = 1      3      6     10     15
```

The inverse operation of `cumsum` is `diff` , which computes the difference between successive elements of the input vector.

```
>> D = diff(C)

D = 2      3      4      5
```

Notice that the output vector is shorter by one than the input vector. As a result, MATLAB’s version of `diff` is not exactly the inverse of `cumsum` . If it were, we would expect `cumsum(diff(X))` to be `X` :

```
>> cumsum(diff(X))

ans = 1      2      3      4
```

But it isn't.

Exercise 8.1

Write a function named `mydiff` that computes the inverse of `cumsum` so that `cumsum(mydiff(X))` and `mydiff(cumsum(X))` both return `X` .

Products and Ratios

The multiplicative version of `cumsum` is `cumprod` , which computes the *cumulative product*. In math notation, that's

$$P_i = \prod_{j=1}^{i} V_j$$

In MATLAB, that looks like

```
>> V = 1:5

V = 1     2     3     4     5

>> P = cumprod(V)

P = 1     2     6    24   120
```

MATLAB doesn't provide the multiplicative version of `diff` , which would be called `ratio` , and which would compute the ratio of successive elements of the input vector.

Exercise 8.2

Write a function named `myratio` that computes the inverse of `cumprod` , so that `cumprod(myratio(X))` and `myratio(cumprod(X))` both return `X` .

You can use a loop, or if you want to be clever, you can take advantage of the fact that $e^{\ln a+\ln b} = ab$.

This page titled 8.1: Functions and Vectors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **8.1: Functions and Vectors** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

8.1.1: Built-in MATLAB Functions of Vectors

Many of MATLAB's built-in functions work with vectors. In fact, any built-in function that makes sense to work with vectors should work with vector inputs and/or outputs. And any built-in function that makes sense to work with matrices should work with matrix inputs and/or outputs.

Here are some common examples of using MATLAB functions that work with vector inputs:

```
x1 = 5
x2 = 0: 2: 8 % [0 2 4 6 8]
m2 = [1 3 5
4 2 -1];

%% sqrt(x)
sqrt(x1) % 2.2361
sqrt(x2) % 0 1.4142 2.0000 2.4495 2.8284
sqrt(m2) % 1.00+0i 1.7321+0i 2.2361+0i
% 2.00+0i 1.4142+0i 0.0000+1.0i

%% rem(n,d) % remainder of n divided by d
rem(x1,3) % 2 (remainder of 5 divided by 3)
rem(x2,3) % 0 2 1 0 2
rem(m2,3) % 1 0 2
% 1 2 -1

mod(x1,3) % 2 (remainder of 5 divided by 3)
mod(x2,3) % 0 2 1 0 2
mod(m2,3) % 1 0 2
% 1 2 2

%% size(x)
size(x1) % 1 1 (1 row, 1 column)
size(x2) % 1 5 (1 row, 5 columns)
size(m2) % 2 2 (2 rows, 3 columns)
[ncols, nrows] = size(m2)
% ncols = 2
% nrows = 3

%% sin(x) in radians
% cos(x), tan(x) work the same way
sin(x1) % -0.9589
sin(x2) % 0 0.9093 -0.7568 -0.2794 0.9894
sin(m2) % 0.8415 0.1411 -0.9589
% -0.7568 0.9093 -0.8415

%% exp(x) = exponential of e^x, (e = 2.7183)
exp(x1) % 148.4132
exp(x2) % 0.0010 0.0074 0.0546 0.4034 2.9810
exp(m2) % 2.7183 20.0855 148.4132
% 54.5982 7.3891 0.3679
```

```
%% log(x) = the natural logarithm (base e)
% In math and physics, this is called ln(x)
% But log(x) = the natural logarithm has been traditional
% in many computer programming languages since the 1960s
log(x1) % 1.6094
log(x2) % -Inf 0.6931 1.3863 1.7918 2.0794
log(m2) % 0+0i 1.098+0i 1.609+0i
% 1.38+0i 0.693+0i 0.000+3.1416i
% In Matlab, the log base 10 function is log10():
log10(x2) % -Inf 0.3010 0.6021 0.7782 0.9031
```

```
%% You can also nest functions, as shown here.
sin(sqrt(0.5)) % 0.6496 MATLAB fist computes sqrt(0.5), then computes the sin() of that result.
```

.

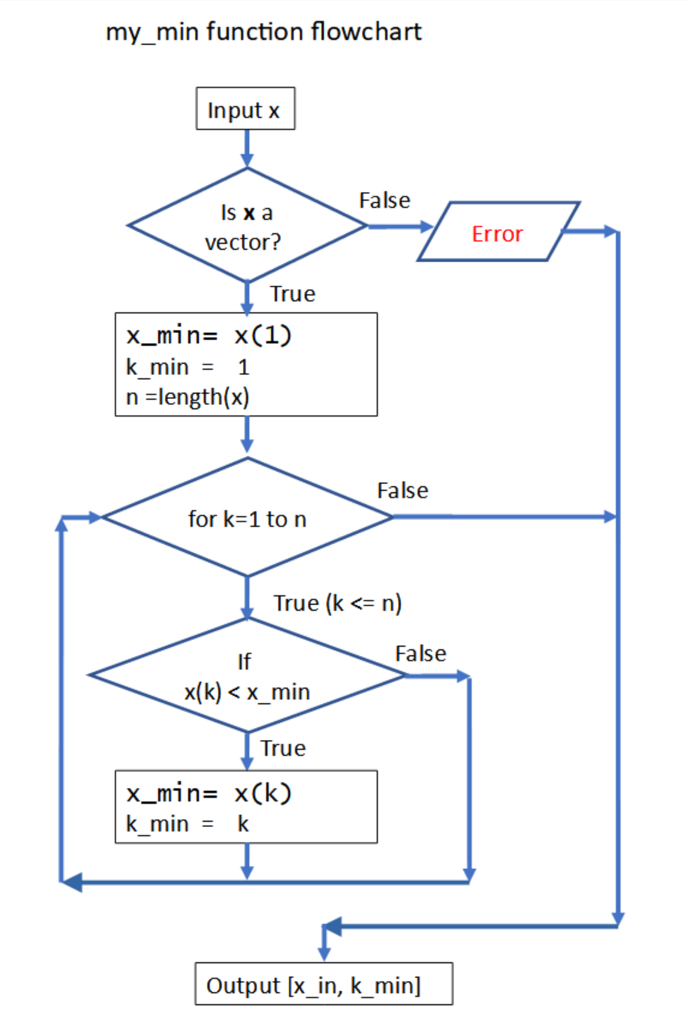
This page titled 8.1.1: Built-in MATLAB Functions of Vectors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

8.1.2: User-Defined Functions of Vectors Examples and Exercises

By Carey Smith

Example 8.1.2.1 my_min_cs


my_min function flowchart
Input x
Is **x** a vector?
False
Error
True
x_min= x(1)
k_min = 1
n =length(x)
for k=1 to n
False
True (k <= n)
If x(k) < x_min
False
True
x_min= x(k)
k_min = k
Output [x_in, k_min]

MATLAB has a built-in minimum function, of course. This example shows how you could write your own minimum function. This kind of logic might be applied to other kinds of searches of a data vector. This algorithm works on a vector input only. It returns both the minimum value and its index.

A key idea is to use the first element of the vector as a candidate for the minimum value, then search for values less that the candidate value. If such a value is found, then the new, smaller value replaces the candidate value .

```
function [x_min, k_min] = my_min(x)
% Input:
% x = data vector (1xn or nx1)
% Outputs:
% x_min = the minimum value of x
% i_min = the index of the minimum value of x
x_min = -inf;
k_min = -1;

% Is x 1-Dimensional?
[m_rows, n_cols] = size(x);
% If both the number of row and
% number of columns are > 1,
% then the vector is 2D
if( (m_rows > 1) & (n_cols > 1) )
    disp('Error: input is not 1-Dimensional');
    return;
end
% If it get this far, then the vector is 1-Dimensional
k = 1; % 1st index into the input vector
x_min = x(k); % Initialize the min value to the first value
k_min = k;
n = length(x);
% This loop compares each value to the current min.
for k= 2:n % Start with k=2, because we initialed to k=1
    if(x(k) < x_min) % Is this value bigger than min so far?
        x_min = x(k); % Reset the min so far to this value
        k_min = k;
    end
end

end % end of function
```

Solution

Try testing it with this input vector:

x = [3,2,3,5,6,4,1,4]
[x_min, idx_min] = my_min(x)

.

? Exercise 8.1.2.1 my_max

Modify the my_min example function to create a function m-file that finds the maximum of a vector input.

Answer

Add texts here. Do not delete this text first.

.

This page titled 8.1.2: User-Defined Functions of Vectors Examples and Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

8.2: Computing with Vectors

In this section, we'll look at two common patterns for working with vectors and connect them to the corresponding ideas from mathematics, existential and universal quantification. And you'll learn about logical vectors, which contain the Boolean values 0 and 1.

Existential Quantification

It's often useful to check the elements of a vector to see if there are any that satisfy a condition. For example, you might want to know if there are any positive elements. In mathematical terms, checking whether something exists is called *existential quantification*, and it's denoted with the symbol ∃, which is pronounced "there exists." For example,

$$\exists x \text{ in } S : x > 0$$

means, "there exists some element x in the set S such that $x > 0$." In MATLAB, it's natural to express this idea with a logical function, like `exists`, that returns `1` if there is such an element and `0` if there is not.

```
function res = exists(X)
    for i=1:length(X)
        if X(i) > 0
            res = 1;
            return
        end
    end
    res = 0;
end
```

We haven't seen the `return` statement before ; it's similar to `break` except that it breaks out of the whole function, not just the loop. That behavior is what we want here because as soon as we find a positive element, we know the answer (it exists!) and we can end the function immediately without looking at the rest of the elements.

If we get to the end of the loop, that means we didn't find what we were looking for, so the result is `0`.

Universal Quantification

Another common operation on vectors is to check whether *all* of the elements satisfy a condition, which is called *universal quantification*, denoted with the symbol ∀ and pronounced "for all." So the expression

$$\forall x \text{ in } S : x > 0$$

means "for all elements x in the set S, $x > 0$."

One way to evaluate this expression in MATLAB is to reduce the problem to existential quantification, that is, to rewrite

$$\forall x \text{ in } S : x > 0$$

to the following:

$$\sim\exists x \text{ in } S : x \leq 0$$

where ~∃ means "does not exist." In other words, checking that all the elements are positive is the same as checking that there are no elements that are nonpositive.

? Exercise 8.3

Write a function named `forall` that takes a vector and returns `1` if all of the elements are positive and `0` if there are any nonpositive elements.

Logical Vectors

When you apply a logical operator to a vector, the result is a *logical vector*: a vector whose elements are the logical values 1 and 0. Let's look at an example:

```
>> V = -3:3

V = -3     -2     -1      0      1      2      3

>> L = V>0

L =  0      0      0      0      1      1      1
```

In this example, `L` is a logical vector whose elements correspond to the elements of `V` . For each positive element of `V` , the corresponding element of `L` is `1` .

Logical vectors can be used like flags to store the state of a condition. And they are often used with the `find` function, which takes a logical vector and returns a vector that contains the indices of the elements that are "true."

Applying `find` to `L` from the example above yields

```
>> find(L)

ans = 5      6      7
```

which indicates that elements 5, 6, and 7 have the value 1.

If there are no "true" elements, the result is an empty vector.

```
>> find(V>10)

ans = Empty matrix: 1x0
```

This example computes the logical vector and passes it as an argument to `find` without assigning it to an intermediate variable. You can read this version abstractly as "find the indices of elements of `V` that are greater than 10."

We can also use `find` to write `exists` more concisely:

```
function res = exists(X)
    L = find(X>0)
    res = length(L) > 0
end
```

? Exercise 8.4

Write a version of `forall` using `find` .

This page titled 8.2: Computing with Vectors is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **8.2: Computing with Vectors** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

8.3: Debugging in Four Acts

When you’re debugging a program, and especially if you’re working on a hard bug, there are four things to try:

Reading

Examine your code, read it back to yourself, and check that it means what you meant to say.

Running

Experiment by making changes and running different versions. Often, if you display the right thing at the right place in the program, the problem becomes obvious, but you might have to invest time building scaffolding.

Ruminating

Take some time to think! What kind of error is it: syntax, runtime, or logical? What information can you get from the error messages or from the output of the program? What kind of error could cause the problem you’re seeing? What did you change last, before the problem appeared?

Retreating

At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode. For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don’t understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call “random walk programming,” which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

The way out is to take more time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break sometimes helps with the thinking. So does talking. If you explain the problem to someone else (or even yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works, and then rebuild.

Beginning programmers are often reluctant to retreat, because they can’t stand to delete a line of code (even if it’s wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in, a little bit at a time.

To summarize, here’s the Eighth Theorem of Debugging:

> *Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.*

This page titled 8.3: Debugging in Four Acts is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **8.3: Debugging in Four Acts** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

8.3.1: Debugging Theorems all in 1 Place

Here are the Theorems of Debugging all in one place:

The First Theorem of Debugging: (1.3: A Glorified Calculator)

- Readable code is debuggable code.

The Second Theorem of Debugging: (1.4: MATLAB User Interface)

- The only thing worse than getting an error message is not getting an error message.

The Third Theorem of Debugging: (2.5: The Fibonacci Sequence)

- Be sure that the code you are running is the code you think you are running.

The Fourth Theorem of Debugging: (2.6: Floating-Point Numbers)

- Error messages tell you where the problem was discovered, not where it was caused.

The Fifth Theorem of Debugging: (3.1: Updating Variables)

- The best way to avoid a bug is to make it impossible.

The Sixth Theorem of Debugging: (3.9: Incremental Development)

- The best kind of debugging is the kind you don’t have to do.

The Seventh Theorem of Debugging: (10.2: Nonlinear Interpolation)

- The worst bugs aren’t in your code; they’re in your head.

The Eighth Theorem of Debugging: (8.3: Debugging in Four Acts)

- Finding a hard bug requires reading, running, ruminating, and sometimes retreating.

If you get stuck on one of these activities, try the others.

This page titled 8.3.1: Debugging Theorems all in 1 Place is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) .

8.4: Chapter Review

This chapter presents patterns for working with vectors, including existential and universal quantification. We learned how to write functions that take vectors as input variables and return vectors as output variables. And we learned about *logical vectors*, which contain the values 1 and 0 to represent true and false.

Some of the functions in this chapter are not idiomatic MATLAB; many of them can be done more simply using built-in MATLAB operators and functions, rather than writing them yourself. But these examples demonstrate concepts you will need to know when you work on more complicated problems.

In the next chapter, we will apply the tools we have learned so far to the central goal of this book, modeling physical systems.

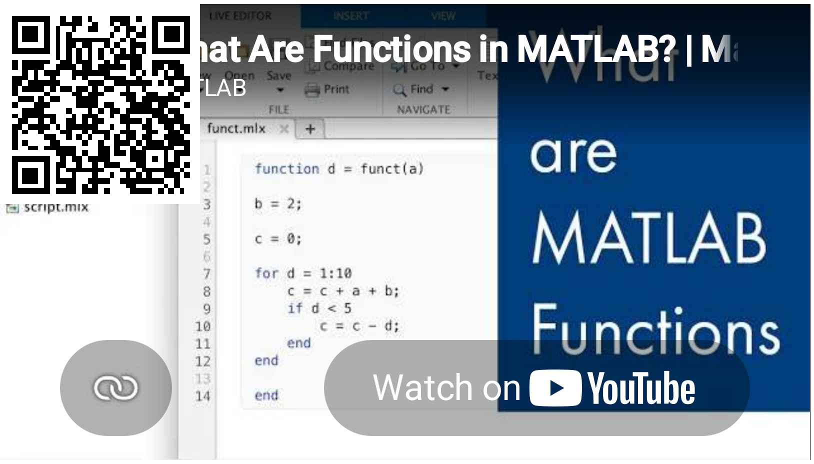
This page titled 8.4: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **8.4: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

8.5: Chapter 8 Function Videos English and Espanol

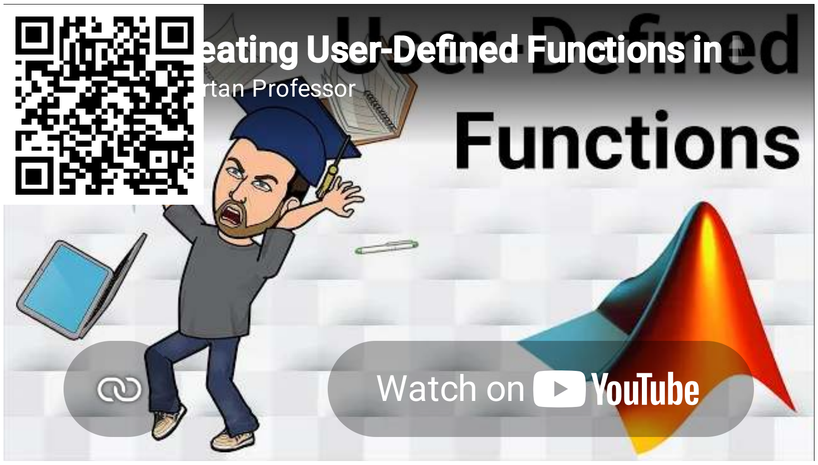
English



```
function d = funct(a)

b = 2;

c = 0;

for d = 1:10
    c = c + a + b;
    if d < 5
        c = c - d;
    end
end

end
```

hat Are Functions in MATLAB? | M
LAB
funct.mlx
script.mlx
What are MATLAB Functions
Watch on YouTube

.


eating User-Defined Functions in
tan Professor
User Defined Functions
Watch on YouTube

.

Debugging in Matlab (Mathworks) (Link to a page that has an embedded video)

https://www.mathworks.com/videos/debugging-in-matlab-97189.html

Matlab Debugging Tutorial (Ilya Mikhelson)

ATLAB Debugging Tutorial
Mikhelson
Watch on YouTube

Matlab Debugger to Walk thru and Understand Code (Link to a page that has an embedded video)

https://www.mathworks.com/videos/using-debugger-to-walk-through-code-97232.html

This is a link to how to debug in Octave:

https://docs.octave.org/interpreter/Entering-Debug-Mode.html

Espanol

Funciones con arreglos y anónimas - 18 - Octave

nciones con arreglos y anónimas
siored
ctave
nciones con
arreglos y
anónimas
Watch on YouTube

This page titled 8.5: Chapter 8 Function Videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

9: MATLAB Random Number and Statistics Functions

9.1: MATLAB Random Number Functions
9.2: MATLAB Statistics Functions
9.3: Statistical Plots
9.4: Random Number Applications
9.5: Chapter 9 Random Number Videos English and Espanol

By Carey A. Smith

This page titled 9: MATLAB Random Number and Statistics Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

9.1: MATLAB Random Number Functions

By Carey A. Smith

Random numbers are useful in MATLAB simulations of real-world signals and events which have uncertainty. There are 3 main MATLAB random number generators:

9.1.1: **randi(IMAX,M,N)** Random Integers

RndInt = **randi(IMAX,M,N)** returns an M-by-N matrix of **uniformly distributed pseudo-random integers** from 1 to IMAX.

Example 1A: Generate a vector of 8 dice rolls:

```
Dice = randi(6,1,8)
```

Example 1B: Generate a 4x3 array of random integers between 0 and 100

RndInt_4_3 = randi(100,4,3)

9.1.2: **rand(M,N)** Random Decimals

RndDec = **rand(M,N)** returns an M-by-N matrix of **uniformly distributed pseudo-random decimal numbers** between 0 and 1.

Example 2A: Generate 8 random decimals between 0 and 1:

```
RndDec2A = rand(1,8)
```

The vector can be scaled to be between any 2 numbers by adding an offset number and multiplying by a scale factor.

Example 2B: Generate 8 random decimals between –1 and 1:

```
RndDec2B = 2*rand(1,8) - 1
```

9.1.3: **randn(M,N)** Normally Distributed Random Numbers

RndNorm = **randn(M,N)** returns an M-by-N matrix of **pseudo-random values drawn from the standard normal distribution**, which has a mean (average) = 0 and a standard deviation = 1.

The standard deviation is a measure of the spread of the data. For a further explanation, watch this video:

“Standard Deviation - Explained and Visualized”, by Jeremy Blitz-Jones

https://www.youtube.com/watch?v=MRqtXL2WX2M

The Normal Distribution is also called the “Gaussian Distribution”.

Example 3A:

```
RndNorm3A = randn(1,5)
```

The result can be scaled to have any mean and standard deviation by adding an offset number and multiply by a scale factor.

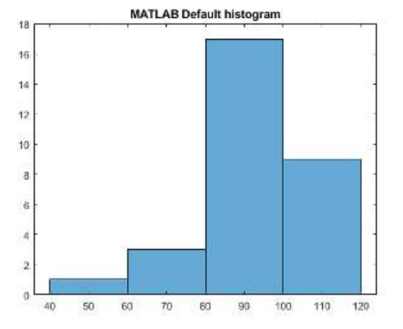
✓ Example 9.1.1 Pulse Rates

```
% Example of randn() for pulse rates
% Generate a sample of 30 random pulse rates
% with a nominal mean = 90
% and a nominal standard deviation = 15.
n = 30
pmean    = 90
pstd_dev = 15
pulse = pmean + pstd_dev*randn(1,n); % This creates a 1xn vector
```

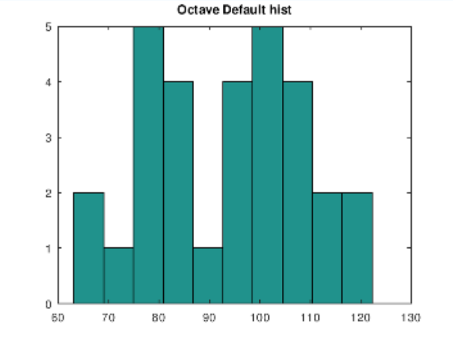
```
% Compute the sample mean and standard deviation.
pulse_mean = mean(pulse)
pulse_std = std(pulse)

%% Create a histogram of the results.
% For MATLAB use the histogram() function
% For Octave, use the older hist() function

%% MATALB Default histogram
figure;
histogram(pulse)
title('MATLAB Default bins')

%% Octave Default histogram

graphics_toolkit("fltk") % Needed for Octave 7.2
figure;
hist(pulse)
title('Octave Default hist')

%% MATLAB Custom histogram
edges = 60 : 10 : 120;
figure;
histogram(pulse, edges)
title('MATLAB Custom bin edges histogram')
```

Solution

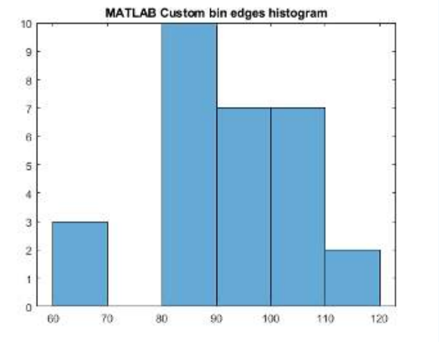
MATLAB Default



Octave:



MATLAB Custom bin edges



.

Example 3C: A radar signal has magnitude 20 with noise of magnitude = 4 (standard deviation)

Received_signal = 20 + 4*randn(1,5)

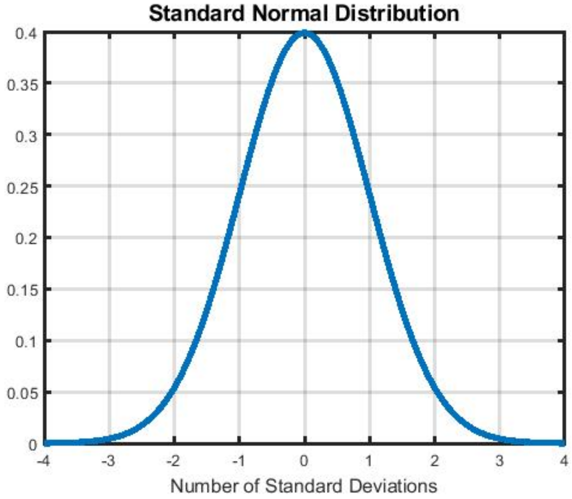


Figure $9.1.1$: Standard Normal Probability Distribution

The total area under the curve = 1.00. Approximately:

68% is between -1 and +1 standard deviations

95% is between -2 and +2 standard deviations

99.7% is between -3 and +3 standard deviations

The normal distribution approximates many things, such as:

- Noise in signals
- Measurement errors
- Clinical trials of medicines and vaccines
- Probability and statistics
- Biological data, such as weights, heights, pulse rates, blood pressure, diabetes, etc.

The “Central Limit Theorem” states that the sum or average of random variables becomes normal as the number of data (the sample size) increases, even if the raw data is not normally distributed.

You may want to watch Mike’s Galton Machine video. It uses ball bearings to build up a normal distribution: https://www.youtube.com/watch?v=VQepRP-f6GA

See also: https://en.Wikipedia.org/wiki/Galton_board

The Standard Normal Probability Distribution plot can be created with this code:

```
% NormalDistribution.m

x = -4:0.01:4;

mu = 0; % mean

std_dev = 1; % standard deviation

y = (1/(std_dev*sqrt(2*pi))) * exp((-1/2)*((x-mu)/std_dev).^2);

figure;

plot(x,y,'LineWidth',4)

grid on;

title('Standard Normal Distribution','FontSize',14)

xlabel('Number of Standard Deviations','FontSize',12)

ax = gca

ax.LineWidth = 2
```

This page titled 9.1: MATLAB Random Number Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

9.2: MATLAB Statistics Functions

By Carey A. Smith

```
% Create these 3 vectors to illustrate the functions in this section:
v1 = [10, 11, 13, 15, 23];
v2 = [10, 11, 13, 15];
v3 = [10, 15, 23, 11, 13]; % Same values as v1, but in a different order
```

9.2.1: mean(x)

```
%% mean(x) = average of a vector or of the columns of a matrix
v1_mean = mean(v1)
% 14.4000
```

9.2.2: std(x)

```
%% std(x) = standard deviation of a vector or of the columns of a matrix
% The standard deviation is a measure of the spread of the data.
v1_std = std(v1)
% 5.1769
```

9.2.3: min(x)

```
%% min(x) = minimum of a vector or the columns of a matrix
v1_min = min(v1)
% 10
v3_min = min(v3) % This demonstrates that the order of the elements doesn't matter for computing the minimum.
% 10
% Both the minimum value and its index can be obtained by specifying 2 outputs:
[v1_min, idx_min] = min(v1)
% v1_min = 10
% idx_min = 1
```

9.2.4: max(x)

```
%% max(x) = maximum of a vector or the columns of a matrix
v1_max = max(v1)
% 23
v3_max = max(v3) % order doesn't matter
% 23
% Both the maximum value and its index can be obtained by specifying 2 outputs:
[v1_max, idx_max] = max(v1)
% v1_max = 23
% idx_max = 5
```

9.2.5: sort(x)

```
v3_sort1 = sort(v3) % Sorts smallest to largest

% 10 11 13 15 23

v3_sort1 = sort(v3,'ascend') % Also sorts smallest to largest

% 10 11 13 15 23

v3_sort3 = sort(v3,'descend') % Sorts largest to smallest

% 23 15 13 11 10
```

Exercise 9.2.1 sort() function assignment

```
%% The following data is from the Camrosa Water District's 2020 Water Quality report.

% (1pt) Define this vector of chloride levels from imported water and from wells:

Chloride = [52 113 122 45 83 133];

% (2 pts) Use this form to determine both the maximum value and the index of the maximum:

[Mx, Idx_max] = max(Chloride) % Returns the index corresponding to the maximum value

% (1 pt) Verify that the index is correct with this line:

Chloride(Idx_max)
```

Answer

Add texts here. Do not delete this text first.

9.2.6: median(x)

```
%% median(x) = middle value of a vector or matrix when the vector is sorted from low to high

v1_median = median(v1)

% 13

% When a vector has an even number of elements, then the median is the mean of the 2 middle values

v2 = [10, 11, 13, 15]

v2_median = median(v2)

% 12
```

9.2.7: Statistics of Matrices

```
M = [ 1  2 5 5;

     -3 -1 0 1;

     -1  3 4 7]

% mean(M) = mean of each column

M_mean_cols = mean(M)
```

```
% -1.0000 1.3333 3.0000 4.3333
% mean(M') = mean of each row
M_mean_rows = mean(M') % mean of the transpose
% 3.2500 -0.7500 3.2500
% 2nd method (better) to compute the mean of the rows:
M_mean_rows = mean(M,2) % mean of the rows or transpose
% 3.2500
% -0.7500
% 3.2500
% To get the mean of the whole matrix use this form:
M_mean_all_1 = mean(M(:)) % The (:) treats all the data of a matrix as a single vector.
% 1.9167
% 2nd form:
M_mean_all_2 = mean(M,'all')
% 1.9167
% The min(), max(), and std() functions for a matrix work similarly the mean() function
```

9.2.8: What Could Go Wrong

Caution: The following code can also be used to compute the mean of a all the elements in a matrix, because mean is a linear function, but it does not work for the std() function, because the standard deviation uses squares and a square root; it is not linear.

```
M_mean_all_3 = mean(mean(M))
% 1.9167
M_std_all = std(M(:)) % correct
% 2.9683
M_std_all_wrong = std(std(M)) % wrong
% 0.4975
```

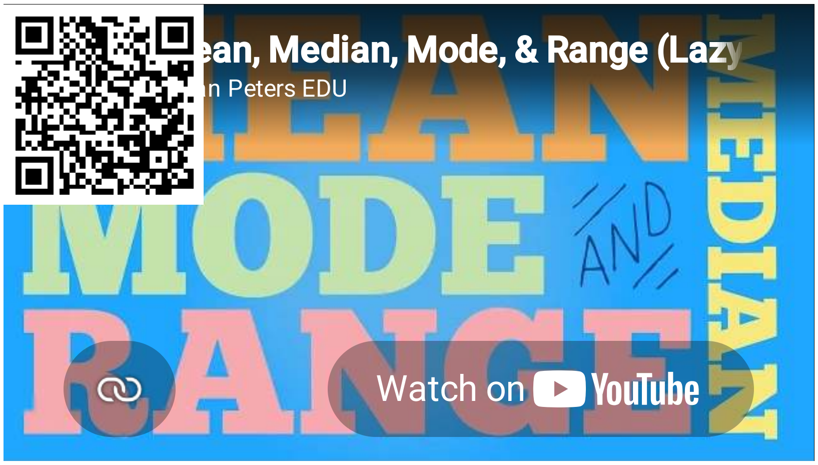
9.2.9: Video Links

Normal distribution standard Deviation - Explained and Visualized (Jeremy Jones)

andard Deviation - Explained and
my Blitz-Jones
STANDARD
DEVIATION
Watch on YouTube

A fun summary of the mean, medium and mode is in this video:

“Mean, Median, Mode, & Range” (Lazy Song Parody), Dylan Peters EDU,


ean, Median, Mode, & Range (Lazy
n Peters EDU
MODE
AND
RANGE
MEDIAN
Watch on YouTube

Exercise 9.2.2 Statistics functions assignment

a. Use the randn() function to generate a vector of 100 normally distributed random data values with a nominal mean of of 50 and a nominal standard deviation of 8.

b. Compute the mean, median, and standard deviation of the random vector.

Since randn() generates a different of random numbers every time it is called. So, rerun your code 4 more times to see how much the mean and standard deviation change. By the “Law of Large Numbers”, the more elements the vector has, then the closer the mean of the generated vector will be to the specified mean.

Answer

Add texts here. Do not delete this text first.

.

This page titled 9.2: MATLAB Statistics Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

9.3: Statistical Plots

By Carey A. Smith

Two types of plots to visualize statistical data are the histogram and the boxchart. To illustrate how to create these, we generate a vector of normally distributed random data with a nominal mean of 50 and a nominal standard deviation of 8.

✓ Example 9.3.1 MATLAB histogram()

This example is only for MATLAB.

The next example is the version for Octave

%% A Histogram is useful for showing the spread of a data set and the frequencies of possible values.

% Examples are ages, blood pressure, home prices, incomes, etc.

% We begin by creating a vector, v4, of random values with a nominal mean and standard deviation:

```
v4_mean0 = 50;

v4_std0 = 8;

v4 = v4_mean0 + v4_std0*randn(1,100);

% Next we compute the actual mean and standard deviation of the created data vector:

v4_mean1 = mean(v4)

v4_std1 = std(v4)

figure

histogram(v4)
grid on;

% Display the mean and standard deviation in the title with 3 significant figures

title(['Histogram: mean= ',num2str(v4_mean1,3),', std.dev.= ',num2str(v4_std1,3)])

% The histogram's bin edges can be set to any custom values you want.

% This code creates a a set of bin edges from -4 standard deviations to +4 standard deviations, in increments of 1 standard deviation.

h_edges = v4_std0*(-4:4) + v4_mean0

figure

histogram(v4, h_edges)

grid on;

title(['Histogram of v4 with custom bin edges, mean= ',num2str(v4_mean1,3),', std.dev.= ',num2str(v4_std1,3)])

% Displays the mean and standard deviation with 3 significant figures.
```

Solution

Add example text here.

.

✓ Example 9.3.2 Octave hist()

This example is only for Octave.

%% For Octave, you need to load a graphics tool kit and use the older function hist(x):

```
graphics_toolkit("qt") % Needed for Octave 7.2
v4_mean0 = 50;
v4_std0 = 8;
v4 = v4_mean0 + v4_std0*randn(1,100);
% Next we compute the actual mean and standard deviation of the created data
vector:
v4_mean1 = mean(v4)
v4_std1 = std(v4)
%% For Octave use the older function hist():
figure
hist(v4)
grid on

% Display the mean and standard deviation in the title with 3 significant figures
title(['hist(): mean= ',num2str(v4_mean1,3),', std.dev.= ',num2str(v4_std1,3)])

%% The histogram's bin centers can be set to any custom values you want.
% This code creates a a set of bin centers from -3.5 standard deviations to +3.5
standard deviations, in increments of 1 standard deviation.
h_centers = v4_std0*(-3.5:3.5) + v4_mean0
figure
hist(v4, h_centers)
grid on;
title(['hist() with custom bin centers, mean= ',num2str(v4_mean1,3),', std.dev.=
',num2str(v4_std1,3)])
% Displays the mean and standard deviation with 3 significant figures.
```

Solution

Add example text here.

.

? Exercise 9.3.1 Mimic 100 dice rolls

Assignment histogram: Mimic 100 dice rolls, plot a histogram, compute the mean and standard deviation

(a) Create a vector with 100 uniform random integers between 1 and 6 with this command:

```
dice = randi(6, 1, 100); % randi generates random integers
```

(b) Set the bin edges to be

```
bin_edges= 0.5 : 1 : 6.5;
```

Plot a histogram of the data using this bin_edges vector.

(c) Compute **mdice** = the mean of the vector **dice**
Compute **sdice =** the standard deviation of the vector **dice**
Write the mean and standard deviation in the title using this line of code:

```
title(['100 Dice Rolls, Mean= ' , num2str(mdice) , ' , Std. Dev.= ' ,
num2str(sdice)]);
```

You may want to repeat this a few times, to see how it varies as the random number vector is regenerated.

Answer

One iteration looks like this:

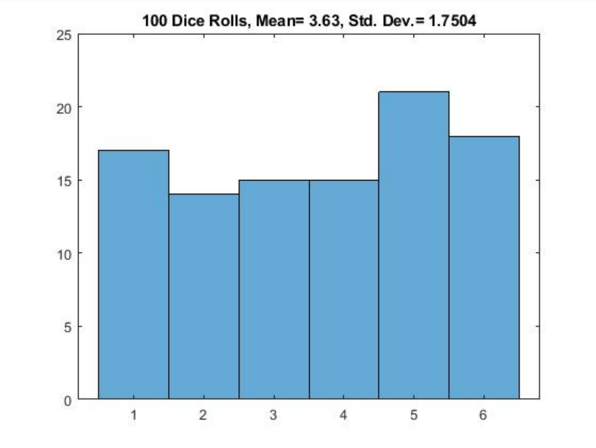


Figure 9.3.1: Historgram Example: Mimic 100 dice rolls

Boxchart (a.k.a. boxplot)

%% A boxchart is another plot of the data.

% It is often used for data that is not normally distributed. It shows the following:

- minimum
- 25th-percentile (1st quartile or Q1)
- median (50th-percentile)
- 75th-percentile (3rd quartile or Q3)
- maximum

✓ Example 9.3.3 boxplot

This example code shows you how to create a MATLAB boxplot.

```
load patients % This is a built-in sample file with various data
n = length(Age) % Number of patients
figure('Name','boxchart() Example')
boxchart(Age)
title('Patient''s Ages')
ylabel('Age (years)')
xlabel([num2str(n),' Patients'])
ylim([20,60])
Age_min = min(Age)
Age_median = median(Age)
Age_max = max(Age)
text(1,(Age_min+1), ['min= ',num2str(Age_min)])
text(1,(Age_median+1),['median= ',num2str(Age_median)])
text(1,(Age_max+1), ['max= ',num2str(Age_max)])
```

Solution

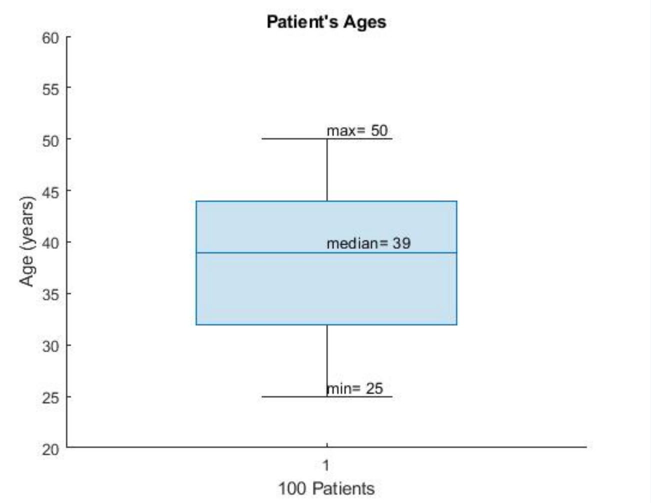


Figure 9.3.2: Boxchart example of patients' ages

Note: Matlab creates vertical boxplots. Books and other software typically create horizontal boxplots. The same information is shown in both types.

This page titled 9.3: Statistical Plots is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

9.4: Random Number Applications

In engineering applications, such as radar or satellite signals, the data has random noise. A process called filtering can be used to reduce the noise. There are multiple types of filters. The following is one example.

✓ Example 9.4.1 Low-Pass Filter with a Step Input

A first-order, low-pass filter smooths noisy data by using a for-loop with the following equation, which sets the output at each time step to be a weighted average of the previous output and the current input:

$y_k = gainy*y_{k-1} + gainx*x_k$

where gainy + gainx = 1.

The bandwidth chosen for the filter sets these gain parameters.

The algorithm steps are:

1. Define a time vector
2. Create a pure step function input
3. Define the parameters for a 1st Order Low-Pass Filter.
4. Apply the 1st Order Low-Pass Filter to the pure step input using a for-loop.
5. Create a 2nd input = step function + noise
6. Apply the 1st Order Low-Pass Filter to the noisy input using a for-loop.

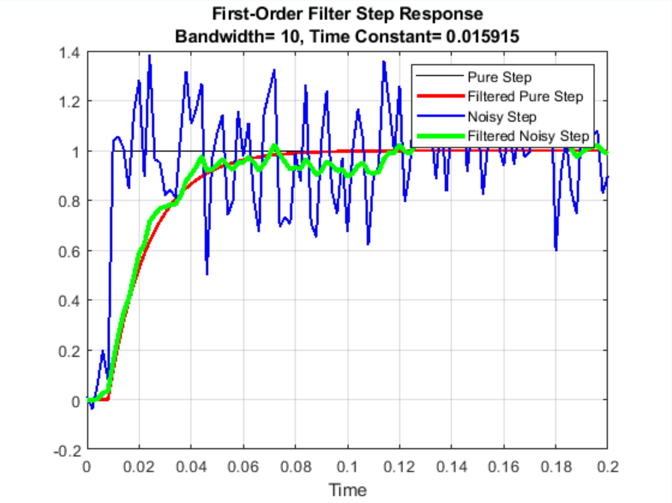
Write a for loop to implement the filter equation.

```
%% Low_Pass_Filter_Step_2023.m
clear all; format compact; format short; clc; close all;
%% a. Define the time and signal vectors
% Create the time vector
dt = 0.002;
t = 0 : dt : 0.4;        %(s)
% Create the step function
s0 = ones(size(t));      % Initialize the step function
s0(1:5) = 0;       % set data to zero before the step occurs.
% Plot the pure signal
figure;
plot(t,s0,'k');
grid on;
hold on; % Plot the other graphs on the same figure

%% b. Add noise to the signal
noise_std = 0.2;  % Standard deviation of the noise
s1 = s0 + noise_std*randn(size(s0)); % signal + noise
plot(t,s1,'bs');  % Plot as blue squares

%% c. Define the parameters for a 1st Order Low-Pass Filter
% A 1st Order Low-Pass Filter smooths the data
% Each iteration's output is a weighted combination
%  of the current measurement + the previous filter output.
bw    = 10                 %(Hz) Bandwidth
tau   = 1/(2*pi*bw)        %(s) Time constant
% The output rises to a factor of (1-(1/e)) of the step in tau seconds
gain_y = exp(-2*pi*bw*dt)
gain_s = 1-gain_y

%% d. Apply the 1st Order Low-Pass Filter
%       to the noisy signal using a for-loop.
y1    = zeros(size(s1));  % initial the output to zeros
t_len = length(t)
for k = 2:t_len
  y1(k) = gain_y*y1(k-1) + gain_s*s1(k);
end
hold on;
plot(t,y1,'go'); % Plot as green circles
title({'First-Order Filter Step Function',...
      ['Bandwidth= ',num2str(bw),', Time Constant= ',num2str(tau)]});
% The { } allows a 2-line title
xlabel('Time');
legend('Pure Signal', 'Noisy Signal', 'Filtered Signal')
```

Low_Pass_Filter_Step.m

Solution

An example of the result for one realization of random noise is shown in this figure.



.

Exercise 9.4.1 Low-Pass Filter with a Sine Wave Input

Modify Low_Pass_Filter_Step.m as follows:

a. Change the input to a sine wave with this code:

```
dt = 0.002; % s
t = 0 : dt : 0.4; % s
```

```
f1 = 10 % Hz
s0 = sin(2*pi*f1*t);
% Don't set any data to zero
```

b. Because the sine wave has both positive values, delete this line: `ylim([0,1.5]);`

c. Change the filter bandwidth from 10 to 20 Hz:

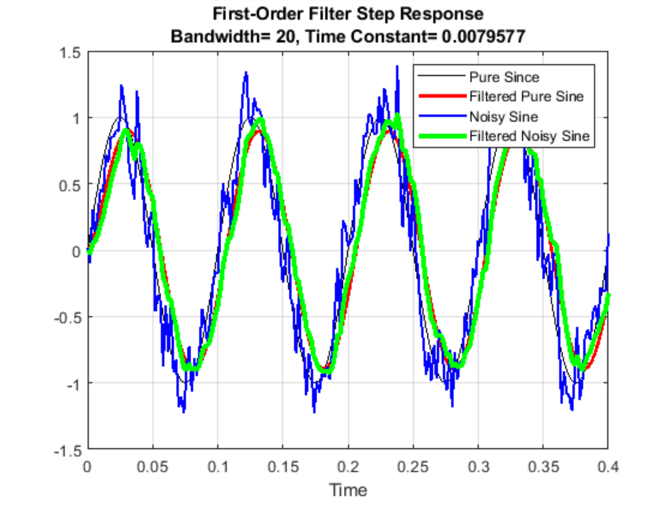
```
bw = 20 %(Hz) Bandwidth
```

d. Change the legend to be this:

```
legend('Pure Since', 'Filtered Pure Sine', 'Noisy Sine','Filtered Noisy Sine')
```

Answer

One noise realization looks like this:



.

This page titled 9.4: Random Number Applications is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

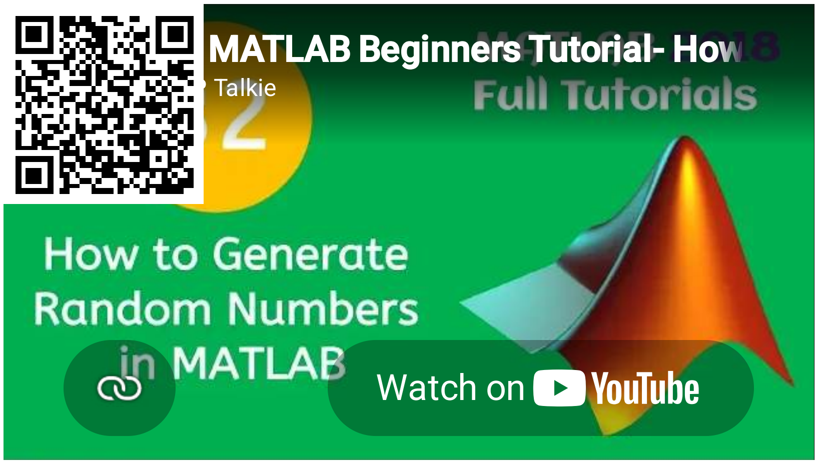
9.5: Chapter 9 Random Number Videos English and Espanol

English

Normal Distribution Demonstration (Bean Machine video)

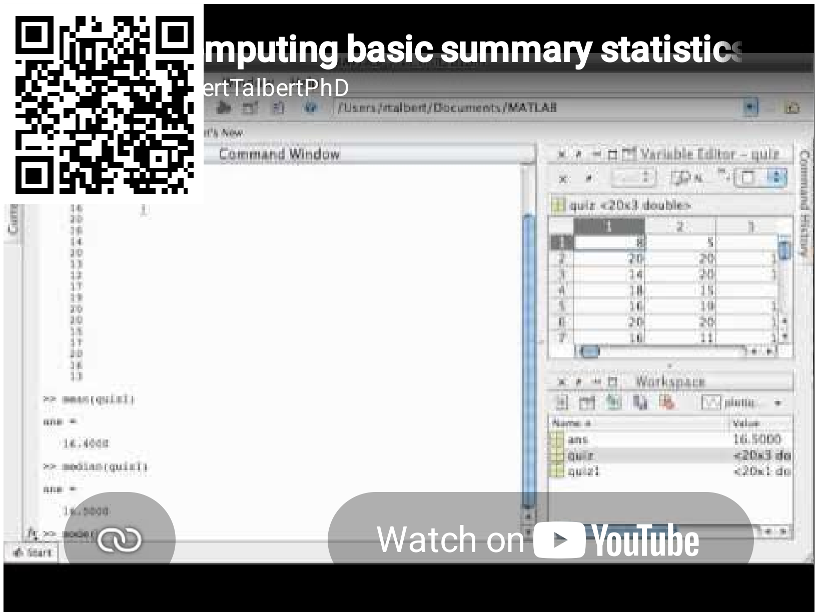
rmal Distribution Demonstration (
Watch on YouTube

.


MATLAB Beginners Tutorial- How
Talkie
Full Tutorials
How to Generate Random Numbers in MATLAB
Watch on YouTube

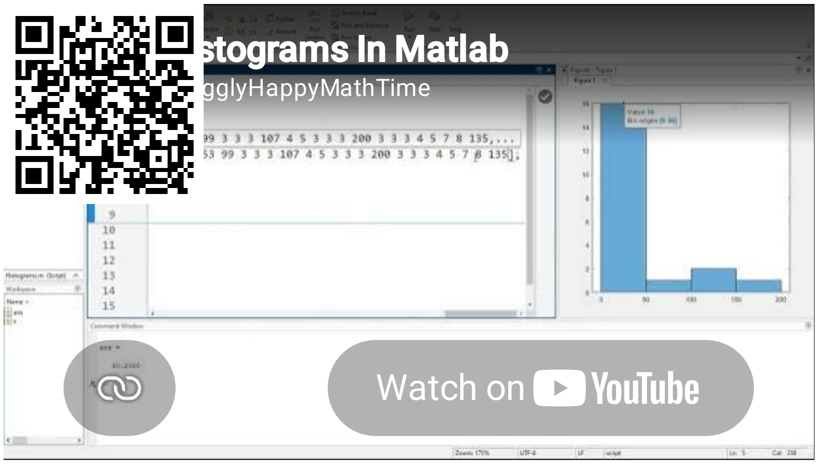
.

mean, std, median, max, min video (RTalbert)


mputing basic summary statistics
ertTalbertPhD
Watch on YouTube

2022 How to Plot a Histogram in MATLAB (Phil Parisi)

w to Plot a Histogram in MATLAB
Parisi
t in MATLAB
MATLAB
PLOTTING
Watch on YouTube


stograms In Matlab
gglyHappyMathTime
Watch on YouTube

.

Espanol

Histograma y Random - 22 - Octave

Histograma y Random - 22 - Octave
siored
Octave
22 Histogramas
Random
Watch on YouTube

This page titled 9.5: Chapter 9 Random Number Videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

10: Interpolation and Curve Fitting

10.1: Linear Interpolation
  10.1.1: Linear Interpolation Problem Set
10.2: Nonlinear Interpolation
10.3: Linear and Polynomial Curve Data Fitting
10.4: Polynomial Fitting Problem Set
10.5: Interactive Fitting Tools
10.6: Chapter 10 Curve Fitting Videos in English and Espanol

Interpolation is estimating the value of a function between known data points.

Curve fitting is estimating the coefficients of a function that approximates the given data points. This is often used when the given data is measured data that is assumed to have some errors. Another name for curve fitting is "regression". (Some people make a distinction between these, based on the assumptions and purposes, but the computation is the same for both.)

This page titled 10: Interpolation and Curve Fitting is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

10.1: Linear Interpolation

This is from Serhat Beyenir, A_brief-introduction-to-engineering-computation-with-matlab-11.8.pdf

Some formatting has been corrected.

Linear interpolation is one of the most common techniques for estimating values between two given data points. For example, when using steam tables, we often have to carry out interpolations. With this technique, we assume that the function between the two points is linear. MATLAB has built-in interpolation functions.

10.1.1: The interp1 Function

Given an x-y table, y_new = interp1(x,y,x_new) interpolates to find a y_new. Consider the following examples:

✓ Example 10.1.1

Example 5.1
To demonstrate how the interp1 function works, let us create an x-y table with the following
commands;

```
x = 0:5;
y = [0,20,60,68,77,110];
```

To tabulate the output, we can use

```
[x',y']
```

The result is

```
ans =
0 0
1 20
2 60
3 68
4 77
5 110
```

Suppose we want to find the corresponding value for 1.5 or interpolate for 1.5.

Using y_new = interp1(x,Y,x_new) syntax, let us type in:

```
y_new=interp1(x,y,1.5)
y_new =
40
```

Solution

Add example text here.

.

✓ Example 10.1.2

The table we created above has only 6 elements in it and suppose we need a more detailed table.
In order to do that, instead of a single new x value, we can define an array of new x values.

The interp1 function returns an array of new y values:

```
new_x = 0:0.2:5;
new_y = interp1(x,y,new_x);
```

Let's display this table

```
[new_x',new_y']
```

The result is

```
ans =
0 0
0.2000 4.0000
0.4000 8.0000
0.6000 12.0000
0.8000 16.0000
1.0000 20.0000
1.2000 28.0000
1.4000 36.0000
1.6000 44.0000
1.8000 52.0000
2.0000 60.0000
2.2000 61.6000
2.4000 63.2000
2.6000 64.8000
2.8000 66.4000
3.0000 68.0000
3.2000 69.8000
3.4000 71.6000
3.6000 73.4000
3.8000 75.2000
4.0000 77.0000
4.2000 83.6000
4.4000 90.2000
4.6000 96.8000
4.8000 103.4000
5.0000 110.0000
```

Solution

Add example text here.

.

✓ Example 10.1.3

Using the table below, find the internal energy of steam at 215 ºC and the temperature if the internal energy is 2600 kJ/kg (use linear interpolation).

First let us enter the temperature and energy values

```
temperature = [100, 150, 200, 250, 300, 400, 500];
energy = [2506.7, 2582.8, 2658.1, 2733.7, 2810.4, 2967.9, 3131.6];
[temperature',energy']
```

This returns

```
ans =
1.0e+003 *
0.1000 2.5067
0.1500 2.5828
0.2000 2.6581
0.2500 2.7337
0.3000 2.8104
0.4000 2.9679
0.5000 3.1316
```

Issue the following for the first question:

```
new_energy = interp1(temperature,energy,215)
```

returns

```
new_energy =
2.6808e+003
```

Now, type in the following for the second question:

```
new_temperature = interp1(energy,temperature,2600)
```

returns

```
new_temperature =
161.4210
```

Solution

Add example text here.

. Note that MATLAB also has a function interp2() for 2-dimensional interpolation. Information about and examples of interp2 can be foun in MATLAB's help.

Summary of Key Points

1. Linear interpolation is a technique for estimating values between two given data points.
2. Problems involving steam tables often require interpolated data.
3. MATLAB has a built-in interpolation function.

This page titled 10.1: Linear Interpolation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Serhat Beyenir.

10.1.1: Linear Interpolation Problem Set

Determine the saturation temperature, specific liquid enthalpy, specific enthalpy of evaporation and specific enthalpy of dry steam at a pressure of 2.04 MPa.

Pressure [MN/m^2]	**Saturation Temperature [C]**	**h_f [kJ/kg]**	**h_{fg} [kJ/kg]**	**h_g [kJ/kg]**
2.1	214.9	920.0	1878.2	2798.2
2.0	212.4	908.6	1888.6	2797.2

Answer

MATLAB solution is as follows; >> pressure=[2.1 2.0]; >> sat_temp=[214.9 212.4]; >> h_f=[920 908.6]; >> h_fg=[1878.2 1888.6]; >> h_g=[2798.2 2797.2]; >> sat_temp_new= interp1(pressure, sat_temp,2.04) sat_temp_new = 213.4000 >> h_f_new=interp1(pressure,h_f,2.04) h_f_new = 913.1600 >> h_fg_new=interp1(pressure,h_fg,2.04) h_fg_new = 1.8844e+003 >> h_g_new=interp1 (pressure,h_g,2.04) h_g_new = 2.7976e+003

The following table gives data for the specific heat as it changes with temperature for a perfect gas. (Data available for download). [1]

Temperature [F]	**Specific Heat [BTU/lbmF]**
25	0.118
50	0.120
75	0.123
100	0.125
125	0.128
150	0.131

Change of specific heat with temperature

Using interp1 function calculate the specific heat for 30 F, 70 F and 145 F.

Answer

MATLAB solution is as follows: >> temperature=[25;50;75;100;125;150] temperature = 25 50 75 100 125 150 >> specific_heat=[.118;.120;.123;.125;.128;.131] specific_heat = 0.1180 0.1200 0.1230 0.1250 0.1280 0.1310 >> specific_heatAt30=interp1(temperature,specific_heat,30) specific_heatAt30 = 0.1184 >> specific_heatAt70=interp1(temperature,specific_heat,70) specific_heatAt70 = 0.1224 >> specific_heatAt145=interp1(temperature,specific_heat,145) specific_heatAt145 = 0.1304

For the problem above, create a more detailed table in which temperature varies between 25 and 150 with 5 F increments and corresponding specific heat values.

Answer

MATLAB solution is as follows: >> new_temperature=25:5:150; >>

new_specific_heat=interp1(temperature,specific_heat,new_temperature);>> [new_temperature',new_specific_heat'] ans = 25.0000 0.1180 30.0000 0.1184 35.0000 0.1188 40.0000 0.1192 45.0000 0.1196 50.0000 0.1200 55.0000 0.1206 60.0000

0.1212 65.0000 0.1218 70.0000 0.1224 75.0000 0.1230 80.0000 0.1234 85.0000 0.1238 90.0000 0.1242 95.0000 0.1246 100.0000 0.1250 105.0000 0.1256 110.0000 0.1262 115.0000 0.1268 120.0000 0.1274 125.0000 0.1280 130.0000 0.1286 135.0000 0.1292 140.0000 0.1298 145.0000 0.1304 150.0000 0.1310

During a 12-hour shift a fuel tank has varying levels due to consumption and transfer pump automatically cutting in and out to maintain a safe fuel level. The following table of fuel tank level versus time (Data available for download) is missing readings for 5 and 9 AM. Using linear interpolation, estimate the fuel level at those times.

Time [hours, AM]	Tank level [m]
1:00	1.5
2:00	1.7
3:00	2.3
4:00	2.9
5:00	?
6:00	2.6
7:00	2.5
8:00	2.3
9:00	?
10:00	2.0
11:00	1.8
12:00	1.3

Fuel Tank Level Versus Time

Answer

>> time=[1 2 3 4 6 7 8 10 11 12]; >> tank_level=[1.5 1.7 2.3 2.9 2.6 2.5 2.3 2.0 1.8 1.3]; >> tank_level_at_5=interp1(time,tank_level,5);tank_level_at_5=2.7500>>tank_level_at_9=interp1(time,tank_level,9) tank_level_at_9 = 2.1500

10.1.1.1: Footnotes

- 1 *Thermodynamics and Heat Power by Kurt C. Rolle, Pearson Prentice Hall. © 2005, (p.19)*

This page titled 10.1.1: Linear Interpolation Problem Set is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Serhat Beyenir via source content that was edited to the style and standards of the LibreTexts platform.

- **1.5.1: Problem Set** by Serhat Beyenir is licensed CC BY 4.0. Original source: https://cnx.org/contents/qTr2bTqz11.8:IZLgFOgb3.

10.2: Nonlinear Interpolation

A smoother interpolation than linear interpolation is called spline interpolation. Originally, a "spline" was a strip of flexible material that could be bent, so that a person could make a smooth curve through data points. The mathematical spline connects the data points with 3rd-order polynomial segments whose 1st and 2nd derivatives are continuous at each point. In MATLAB/Ocatave "spline" is an option with the interp1() function. The next example show how to use it and what the result is.

✓ Example 10.2.1 Spline interpolation

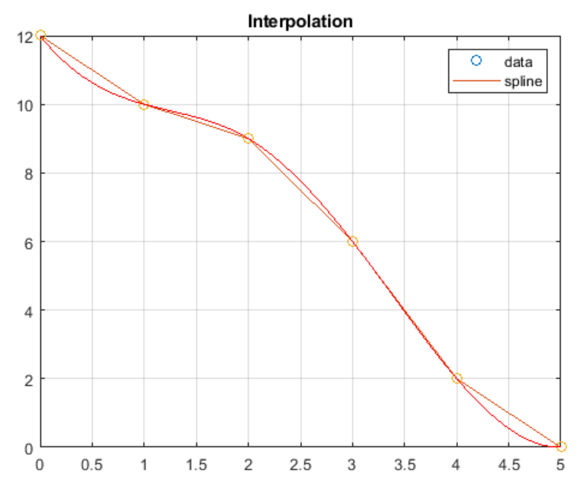
This data is from Seimers example 40. It shows that the spline is smoother than the linear interpolation.

```
x1 = 0:5;
y1 = [12,10,9,6,2,0];
figure;
plot(x1, y1, 'o')
grid on
```

```
%% Interpolate with linear segments
x2 = 0: 0.1: 5; % Point spacing for plotting the fitted curve
% The default interpolation method is 'linear':
y2 = interp1(x1, y1, x2); % Compute the curve
hold on;
plot(x2, y2)
title('Interpolation with lines between the data points')
```

```
%% Interpolate with a smooth spline curve
y3 = interp1(x1, y1, x2, 'spline'); % Compute the curve with the 'spline' option
hold on;
plot(x2,y3,'r')
title('Interpolation')
legend('data', 'spline')
```

Solution



.

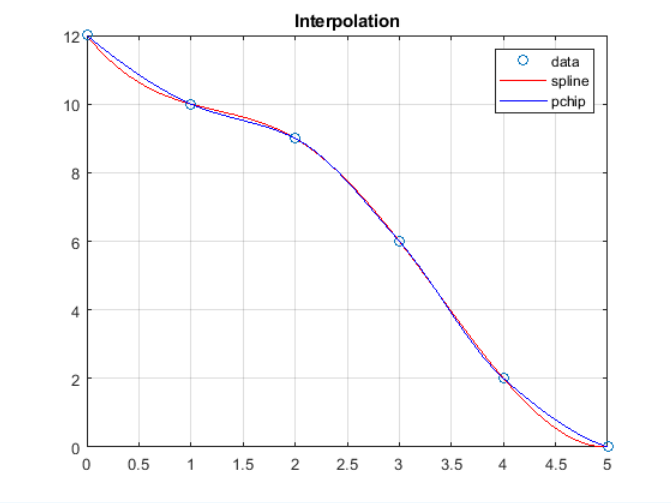
Another 3rd-order interpolation method in MATLAB is "pchip". The difference is the assumptions on the derivatives in the 2 end segments. The pchip is a newer algorithm and is often preferred over the spline.

Example 10.2.2 Pchip Interpolation

Pchip interpolation is compared to spline interpolation in this example, using the same data and spline as in the previous example.

```
figure;
plot(x1,y1,'o')
grid on
y3 = interp1(x1, y1, x2, 'spline'); % Compute the curve
hold on;
plot(x2,y3,'r')
title('Interpolation')

%% Interpolate with a smooth pchip fit
y3 = interp1(x1, y1, x2, 'pchip'); % Compute the curve with the 'pchip' option
plot(x2,y3,'b')
title('Interpolation')
legend('data', 'spline', 'pchip')
% 'pchip' and 'spline' only differ in the 2 end segments
```

Solution



.

Exercise 10.2.1 Nonlinear Interpolation Exercise1

Create an m-file script. Define this a, y data set:

```
x0 = -10:3:10;
y0 = -5 +2*x0 + 0.9*x0.^2 - 0.3*x0.^3;
```

Open a figure and plot this data using the 'o' marker. (No line)

Use hold on.

Define these x-points at which to compute the interpolated curves:

```
x2 = -10:0.5 : 10;
```

Interpolate the data from the x, y data to the x2 values with linear interpolation. Plot the result on the same figure.

Interpolate the data from the x, y data to the x2 values with spline interpolation. Plot the result on the same figure.

Interpolate the data from the x, y data to the x2 values with pchip interpolation. Plot the result on the same figure.

Add a legend and a title.

Answer

The solution is not given here.

There is also the interp2() function for doing interpolation on a 2-dimensional array of data.

? Exercise 10.2.2 Nonlinear Interpolation Exercise2

Construct this data set:

x0 = -10:3:10;
y0 = -5 + 10*x0 - 2*x0.^3 + 0.1*x0.^4;

Open a figure.

Do the following interpolations and plot them on the same figure: Linear, spline, and pchip.

Add a legend.

Answer

The solution is not given here.

.

? Exercise 10.2.3 Weather Temperature Interpolation

Go to weather.gov. Enter your zip code. Click on "3 Day History" (under "More Information") and find the temperatures for a recent day for every 3 hours from 6:00 to 21:00 (9 pm).

Open a new figure. Create and plot temperature vs. time (use 24 hour time) with the 's' marker. % (The x-axis is time, the y-axis is temperature.)

Add hold on to your .m file.

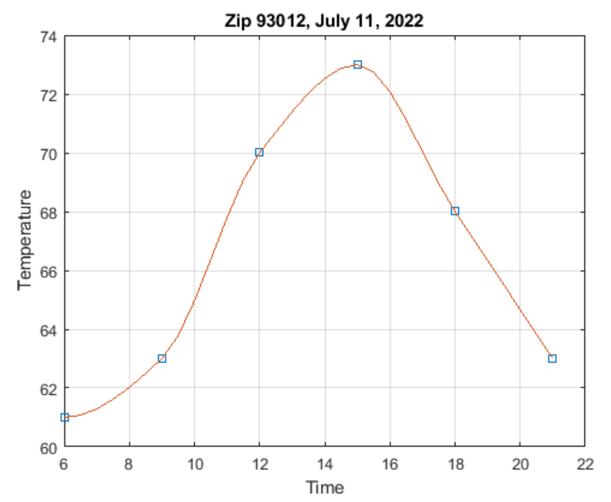
Create a new time vector from 6 to 21 with increments of 0.5 hours. Compute the pchip interpolation for this new time vector and plot it as a curve on the same plot.

Put a title with the zip code and the date of the data.

Label the x and y axes. Turn the grid on.

Answer

This is an example. Use your own data. Your data will be different.



..

This page titled 10.2: Nonlinear Interpolation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Serhat Beyenir.

10.3: Linear and Polynomial Curve Data Fitting

10.3.1: Linear Data Fitting

Interpolation is appropriate when you have a limited amount of accurate data and want to estimate the value of a function between the given data points.

Linear data fitting is appropriate when you have data with random errors, but you believe that the relation between x & y should be approximately linear. In this case, you do not want to connect the data points with lines using interpolation. Rather you want to calculated the best-fit line to the data.This best-fit line will not go through each data point. Rather, it will be a straight line that minimizes the distance between the data points and the line. The approach finds a line that minimizes the sum of the squares of the differences between the data y-values and the y values predicted by y(k) = m*x(k) + b. This is know as the Least-Squares method. (For more information, you can read https://en.wikipedia.org/wiki/Least_squares.) Linear data fitting is also called "linear regression".)

For example, the physics formula relating voltage (V), current (I) and resistance (R) of an electrical component is:

V = I*R

You may want to estimate the resistance of a component by measuring the current through the component and the voltage across the positive and negative connections of the component.

Another example is that you have multiple factors affecting a measurement, but you want a linear approximation for the relation between 2 particular variables.

The main functions for finding the linear approximation are `polyfit()` and `polyval()` . Assuming that your data are in vectors x and y, the steps are:

Step 1: Compute the coefficients using the polyfit() function:

```
coeffs=polyfit(x,y,1);
```

Step 2: Use the computed coefficients to compute the y values of the line at the x data points using the polyval() function.

```
>besty=coeffs(1)*x+coeffs(2);
```

This process is illustrated in the example below.

✓ Example 10.3.1 Linear Fit for data with measurement errors

This example begins by creating linear data with radom errors.

```
x0 = 0:1:10;
y0 = 2*x0;
% Add measurement noise
y1 = y0 + 2*randn(size(y0))
```

One run of this code produced these y1 values:

```
y1  = [1.0759,   4.4454,   4.7333,  7.5520,   9.0903,   9.5448,  12.4408,   12.8191,
17.3326, 16.2794,  18.3967]
```

```
figure;   % Plot these data
hold on;
plot(x0,y1,'ko');
grid on;
```

% Step 1: Compute the coefficients using the polyfit() function:

```
coef1 = polyfit(x0,y1,1)
```

```
% coef1 = [1.6875    1.8999]
```

% Step 2: Use the computed coefficients to compute the y values of the line at the x data points using the polyval() function.

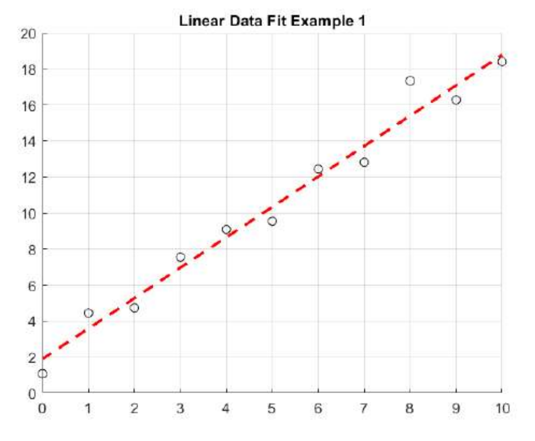
```
y2b = polyval(coef1,x0);
```

% Plot the best fit line on the same plat as the original data

```
plot(x0,y2b,'r--','LineWidth',2);
title('Linear Data Fit Example 1')
```

The plot shows that the line computed by the linear fit averages out most of the radom errors ("noise") in the original data.

Solution



Add example text here.

The next example shows a linear fit to data influenced by multiple factors.

Example 10.3.2 Linear Data Fitting for Vehicle Fuel Economy

This example computes a linear fit to fuel economy vs. vehicle weight.

In this example, fuel economy is the gallons of fuel used per 100 miles.

Factors that affect fuel efficiency include weight, size of the engine, the coefficient of drag, and the friction in all the moving parts, so we don't expect a line to be a perfect fit for fuel efficiency vs. weight. But, it is useful to see the trend between these 2 variables.

% Linear fit to car fuel economy. References for 2024 vehicles
% https://www.fueleconomy.gov/feg/Find...tion=sbsSelect
% https://www.chevrolet.com/

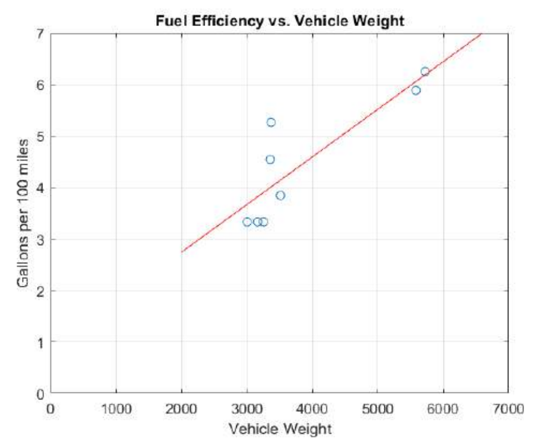
% Variables:
% w = weight
% mpg = miles per gallon
% gp100m = gallons/100 miles
%
% Trailblazer

```
w(1) = 3252;
mpg(1) =   30;
gp100m(1) = 100/mpg(1)
```

% Equinox

```
w(2) = 3512;
mpg(2) =   26;
gp100m(2) = 100/mpg(2)


% Malibu (2023)
w(3) = 3161;
mpg(3) =   30;
gp100m(3) = 100/mpg(3)

% Suburban
w(4) = 5723;
mpg(4) =   16;
gp100m(4) = 100/mpg(4)


% Tahoe
w(5) = 5580;
mpg(5) =   17;
gp100m(5) = 100/mpg(5)

% Chevrolet Camaro, 6-cyl, 3.6 L
w(6) = 3351;
mpg(6) = 22;
gp100m(6) = 100/mpg(6)

% Chevrolet Corvette Stingray
w(7) = 3366;
mpg(7) = 19;
gp100m(7) = 100/mpg(7)

% Chevrolet Trax 1.2 L
w(8) = 3000; % Approximate
mpg(8) = 30;
gp100m(8) = 100/mpg(8)

%% Plot
figure;
plot(w,gp100m,'o')
xlim([0,7000])
ylim([0,7])
grid on;

%% Linear fit
coeffs_eff = polyfit(w, gp100m, 1)
w2 = 2000 : 500: 7000;
eff_fit2 = polyval(coeffs_eff, w2);
hold on;
plot(w2, eff_fit2,'-r');
xlabel('Vehicle Weight')
ylabel('Gallons per 100 miles')
title('Fuel Efficiency vs. Vehicle Weight')
```

Solution



Add example text here.

.

10.3.2: Polynomial Curve Fitting

Sometimes the relation of the data is non-linear. In this case, a polynomial file is better than a linear fit.

The following example is from Troy Siemers (). **9.1: Curve Fitting** by Troy Siemers is licensed CC BY-NC 3.0. Original source:

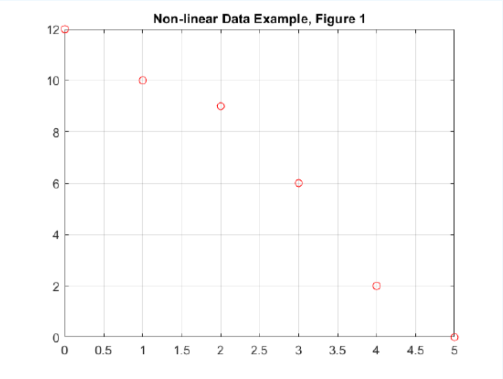
https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

First we show the data with a linear fit.

✓ Example 10.3.2 Curve Fit to Non-Linear Data

```
x=0:5;y=[12,10,9,6,2,0];
```

```
figure
plot(x,y,'o')
```



%% Compute a linear fit to the data

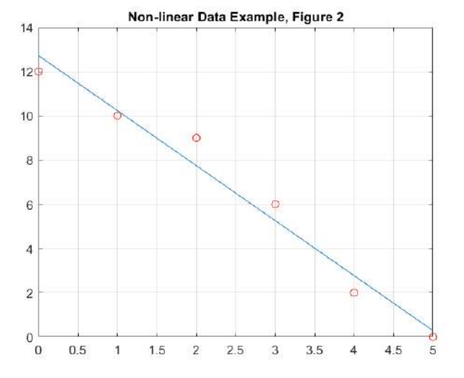
```
 coeffs1 = polyfit(x,y,1);
```

% The command polyfit returns the matrix [-2.4857 12.7143]
% This is the slope and y-intercept of the line of best fit.
% Compute and plot the best-fit line

```
 besty1  = coeffs1(1)*x+coeffs1(2);
```

% Plot the best-fit line on the same graph

```
hold on;
plot(x,besty1)
```



Looking closely at the data, we can see that it curves down. We can compute and plot 2nd-order and 5th-order curve fits.

```
coeffs2 = polyfit(x,y,2);
```

This time, we will use the function polyval(), which is simpler for higher-order curves.

```
x2 = 0 : 0.25 : 5;
y2=polyval(coeffs2,x2);
plot(x2, y2, '+g','Linewidth',3)
title('Non-linear Data Example, 2nd Order Curve Fit')
```

We see that the 2nd-order curve fit is a somewhat better fit to the data.

The polynomial equation can be written on the figure using the text() function.

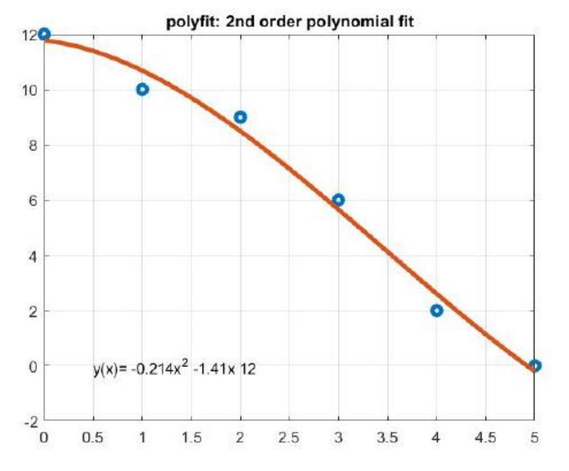
First, we need to create the character string that we want to display with this code:

```
poly_str = ['y(x)= ', num2str(coeffs2(1),3), 'x^2 '...
                      num2str(coeffs2(2),3), 'x '...
                      num2str(coeffs2(3),3)]
```

Note: `num2str(coeffs2(3),3)` converts the 3rd element of coeffs2 to a character string with 3 significant figures.

Then, put the string on the figure with the text() function, starting at point x=0.5, y = 0:

```
text(0.5, 0, poly_str)
```



We can also do a 5th-order fit to the data.

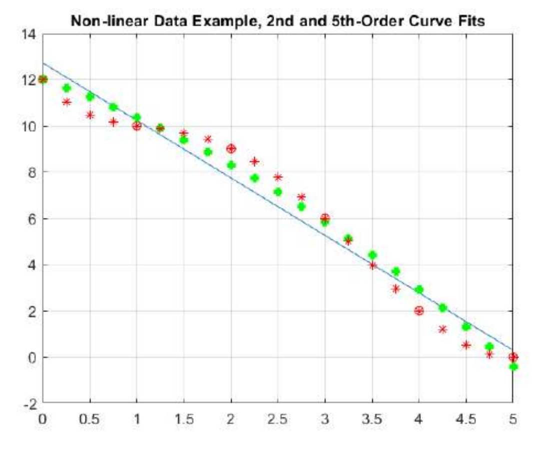
```
coeffs5 = polyfit(x,y,5)
y5 = polyval(coeffs5,x2);
plot(x2, y5, '*r')
title('Non-linear Data Example, 2nd and 5th-Order Curve Fits')
```

Since there are 6 data points, a 5th-order data fit goes through every every point. This might seem like a good thing, but it is likely bogus, unless we have a good reason to believe that the data really is 5th-order. This is known as "over fitting". A rule-of-thumb is that you should have at least 2*n data point for an nth-order curve fit. See these links for details, potential problems, and alternatives:

https://en.wikipedia.org/wiki/Curve_fitting

This Mathworks link explains the peril of extrapolation (plotting the curve for x-values beyond the range of the original data x-values.

https://www.mathworks.com/help/matlab/ref/polyfit.html



Solution

Add example text here.

10.3.3: Fitting by Transforming the Data

Sometimes physical data is measured data as a function of time.

This data in this example is temperature of water as a function of time. The water starts at 100 degrees C and cools off over time.

Both 2nd-order and 3rd-order polynomials are fitted to the data. Both fit well over the first 80 seconds of data. But the 2nd-order fit increases at the end, implying that the water started to get hotter, instead of continuing to cool off. So, the 3rd order curve makes more physical sense.

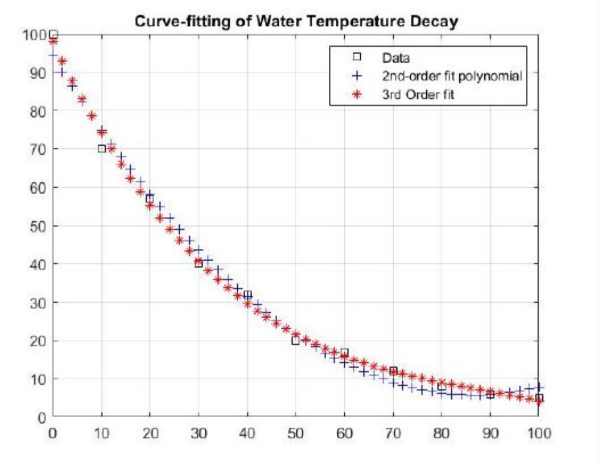
The nex section demonstate show a 5th-order polynomial curve fit to the data in section 10.3

✓ Example 10.3.1 Polyvnomial Curve Fit of Water Temperature vs. Time

```
% Measured data for cooling of hot water vs. time
time1 = 0 : 10: 100;
temperature_t0 = 100 % degrees C
temp0 = temperature_t0*exp(-0.03*time1)
temperature_measured = [100, 70, 57, 40, 32, 20, 17, 12, 8, 6, 5]
figure;
plot(time1, temperature_measured,'sk');
grid on;
hold on;
% 2nd order fit
coeffs_temp2 = polyfit(time1, temperature_measured, 2)
time2 = 0 : 2: 100;
temp_fit2 = polyval(coeffs_temp2, time2);
plot(time2, temp_fit2,'+b');
% 3rd-order fit
coeffs_temp3 = polyfit(time1, temperature_measured, 3)
temp_fit3 = polyval(coeffs_temp3, time2);
plot(time2, temp_fit3,'*r');

title('Curve-fitting of Water Temperature Decay')

legend('Data', '2nd-order fit polynomial', '3rd Order fit')
```

Solution

The plot for this example is shown here:



.

10.3.4: Exponential Curve-Fitting

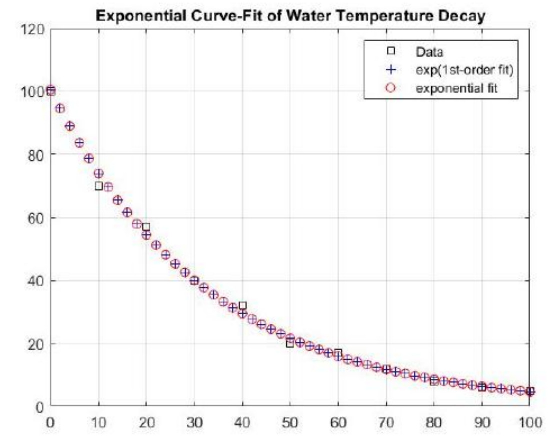
This section and the previous sections show how to do linear and polynomial curve-fitting. Sometimes we expect an exponential function to fit the data batter than a polynomial. An exponential curve is a better fit for many physical phenomena. Often the data can be transformed to be approximately linear. Then a linear fit can be computed and the result can then be transformed back to the original space as an exponential curve. The following is an example of how to do that.

✓ Example 10.3.2 Exponential Curve Fit

```
% Exponential Curve-Fit by Transforming the Data
% Measured data for cooling of hot water vs. time
time1 = 0 : 10: 100;
temperature_t0 = 100 % degrees C
temp0 = temperature_t0*exp(-0.03*time1)
temperature_measured = [100, 70, 57, 40, 32, 20, 17, 12, 8, 6, 5]
figure;
plot(time1, temperature_measured,'sk');
grid on;
hold on;
% We expect the temperature decay to be exponential
% By computing the natural logarithm, the data becomes linear:
log_temperature = log(temperature_measured);
% Linear fit to the logarithm of the data:
log_coeffs_temp1 = polyfit(time1, log_temperature, 1);
log_temp_fit1 = polyval(log_coeffs_temp1, time2);
figure;
plot(time1, log_temperature,'sk');
grid on;
hold on;
plot(time2, log_temp_fit1,'+b');
title('Linear-fit of the Logarithm of Water Temperature')
legend('Data', 'Linear fit')
% Transform back to original space
figure;
plot(time1, temperature_measured,'sk');
grid on;
hold on;
temp_logfit1 = exp(log_temp_fit1);
plot(time2, temp_logfit1,'+b');
% Convert linear fit to an exponential function
temp_fit1 = exp(log_coeffs_temp1(2))*exp(log_coeffs_temp1(1)*time2);
plot(time2, temp_fit1,'or');
title('Exponential Curve-Fit of Water Temperature Decay')
legend('Data', 'exp(1st-order fit)', 'exponential fit')
```

Solution

This plot shows that the exponential fitting process works.



.

This page titled 10.3: Linear and Polynomial Curve Data Fitting is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

- **9.1: Curve Fitting** by Troy Siemers is licensed CC BY-NC 3.0. Original source: https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

10.4: Polynomial Fitting Problem Set

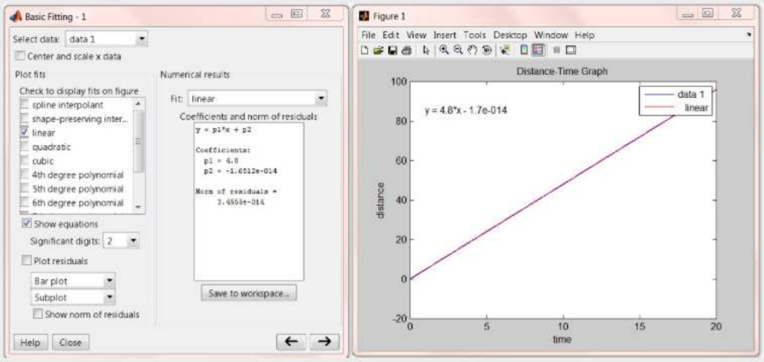
Using the following experimental values [1], plot a distance-time graph and determine the equation, relating the distance and time for a moving object.

Distance [m]	Time [s]
0	0
24	5
48	10
72	15
96	20

Experimental data.

Answer

Data can be entered as follows: distance=[0 24 48 72 96]; time=[0 5 10 15 20]; we can now plot the data by typing in plot(time,distance);title('Distance-Time Graph');xlabel('time');ylabel('distance'); at the MATLAB prompt. The following plot is generated, select Tools > Basic Fitting:



Select data: data 1

☐ Center and scale x data

☑ linear

☑ Show equations

Significant digits: 2

☐ Plot residuals

Fit: linear

```
y = p1*x + p2

Coefficients:
  p1 = 4.8
  p2 = -1.6512e-014

Norm of residuals =
     3.4555e-014
```

Basic Fitting - 1
Plot fits
Check to display fits on figure
☐ spline interpolant
☐ shape-preserving inter...
☑ linear
☐ quadratic
☐ cubic
☐ 4th degree polynomial
☐ 5th degree polynomial
☐ 6th degree polynomial
Numerical results
Coefficients and norm of residuals
Save to workspace...
Bar plot
Subplot
☐ Show norm of residuals
Help
Close
←
→
Figure 1
File Edit View Insert Tools Desktop Window Help
Distance-Time Graph
y = 4.8*x - 1.7e-014
data 1
linear
distance
time
100
80
60
40
20
0
-20
0
5
10
15
20

As shown above, the relationship between distance and time is:

$y = 4.8x - 1.7 \times 10^{-14}$

or

Distance $= 4.8$ Time -1.7×10^{-14}

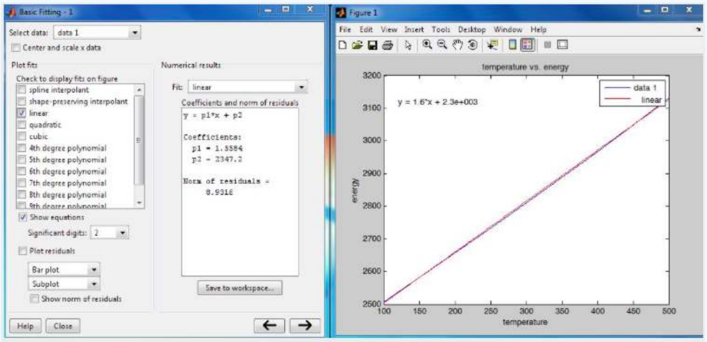
Using the data set below, determine the relationship between temperature and internal energy.

Temperature [C]	Internal Energy [kJ/kg]
100	2506.7
150	2582.8
200	2658.1
250	2733.7
300	2810.4
400	2967.9
500	3131.6

An extract from Steam Tables

Answer

Data can be entered as follows:temperature = [100, 150, 200, 250, 300, 400, 500]; energy = [2506.7, 2582.8, 2658.1, 2733.7, 2810.4, 2967.9, 3131.6]; we can now plot the data by typing in plot(temperature,energy);title('temperature vs. energy');xlabel('temperature');ylabel('energy'); at the MATLAB prompt. The following plot is generated, select Tools > Basic Fitting:


Basic Fitting - 1
Center and scale x data
Plot fits
Check to display fits on figure
spline interpolant
shape-preserving interpolant
quadratic
cubic
4th degree polynomial
5th degree polynomial
6th degree polynomial
7th degree polynomial
8th degree polynomial
9th degree polynomial
Plot residuals
Bar plot
Subplot
Show norm of residuals
Help
Close
Numerical results
Coefficients and norm of residuals
Save to workspace...
Figure 1
File Edit View Insert Tools Desktop Window Help
temperature vs. energy
y = 1.6*x + 2.3e+003
data 1
linear
energy
temperature
3200
3100
3000
2900
2800
2700
2600
2500
100
150
200
250
300
350
400
450
500

Select data: data 1

☑ linear

☑ Show equations

Significant digits: 2

Fit: linear

```
y = p1*x + p2

Coefficients:
  p1 = 1.5584
  p2 = 2347.2

Norm of residuals =
     8.9316
```

As shown above, the relationship between temperature and internal energy is:

$y = 1.6x + 2347.2$

or

internal energy $= 1.6$ temperature $+2347.2$

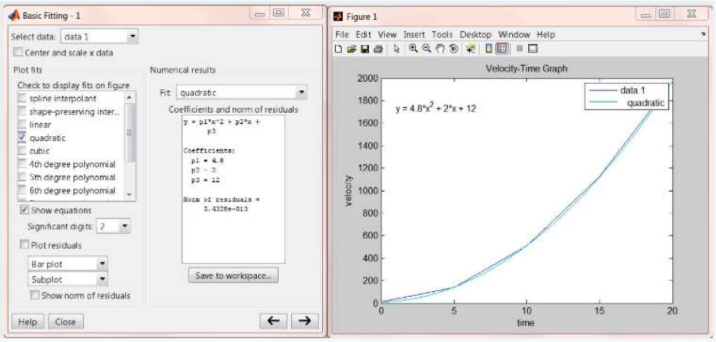
Using the following experimental values [2], plot a velocity-time graph and determine the equation, relating the velocity and time for a moving object.

Velocity [m/s]	Time [s]
12	0
142	5
512	10
1122	15
1972	20

Experimental data.

Answer

Data can be entered as follows: velocity=[12 142 512 1122 1972]; time=[0 5 10 15 20]; we can now plot the data by typing in plot(time,velocity);title('Velocity-Time Graph');xlabel('time');ylabel('velocity'); at the MATLAB prompt. The following plot is generated, select Tools > Basic Fitting, notice that we are choosing the quadratic option this time:


Basic Fitting - 1
Select data: data 1
☐ Center and scale x data
Plot fits
Check to display fits on figure
☐ spline interpolant
☐ shape-preserving inter...
☐ linear
☑ quadratic
☐ cubic
☐ 4th degree polynomial
☐ 5th degree polynomial
☐ 6th degree polynomial
☑ Show equations
Significant digits: 2
☐ Plot residuals
Bar plot
Subplot
☐ Show norm of residuals
Help
Close
Numerical results
Fit: quadratic
Coefficients and norm of residuals
Save to workspace...
Figure 1
File Edit View Insert Tools Desktop Window Help
Velocity-Time Graph
y = 4.8*x^2 + 2*x + 12
data 1
quadratic
velocity
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
5
10
15
20
time

```
y = p1*x^2 + p2*x +
     p3

Coefficients:
  p1 = 4.8
  p2 = 2
  p3 = 12

Norm of residuals =
     5.4329e-013
```

As shown above, the relationship between velocity and time is:

$y = 4.8x^2 + 2x + 12$

10.4.1: Footnotes

- 1 *Engineering Science by E. Hughes and C. Hughes, Longman © 1994, (p. 375)*
- 2 *Engineering Science by E. Hughes and C. Hughes, Longman © 1994, (p. 375)*

This page titled 10.4: Polynomial Fitting Problem Set is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Serhat Beyenir via source content that was edited to the style and standards of the LibreTexts platform.

- **1.7.1: Problem Set** by Serhat Beyenir is licensed CC BY 4.0. Original source: https://cnx.org/contents/qTr2bTqz11.8:IZLgFOgb3.

10.5: Interactive Fitting Tools

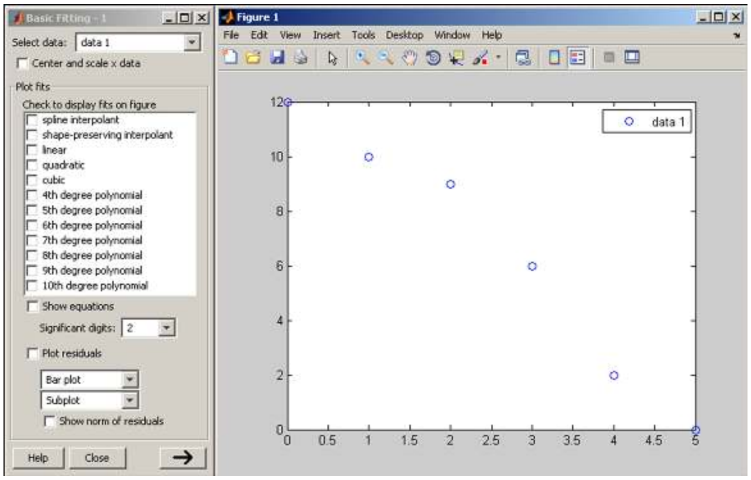
If you aren't sure about the fit you want, you can use the interactive fitting tools. We start again by closing any figure windows, clearing out the variables and clearing the Command Window. A short cut to do this could be:

```
>> close all,clear,clc
```

Re-enter the data.

```
>> x=0:5;
>> y=[12,10,9,6,2,0];
>> plot(x,y,'o')
```

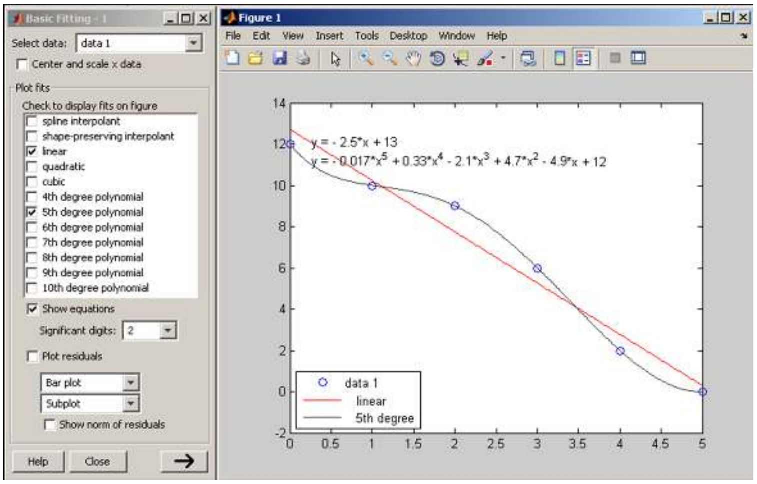
On the figure screen, go to the menu item Tools > Basic Fitting. This will launch a new window:


Basic Fitting - 1
Figure 1
File Edit View Insert Tools Desktop Window Help
Center and scale x data
Plot fits
Check to display fits on figure
☐ spline interpolant
☐ shape-preserving interpolant
☐ linear
☐ quadratic
☐ cubic
☐ 4th degree polynomial
☐ 5th degree polynomial
☐ 6th degree polynomial
☐ 7th degree polynomial
☐ 8th degree polynomial
☐ 9th degree polynomial
☐ 10th degree polynomial
☐ Show equations
☐ Plot residuals
Bar plot
Subplot
☐ Show norm of residuals
Help
Close
→
data 1
0
2
4
6
8
10
12
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

Select data: data 1

Significant digits: 2

Now you simply have to select the polynomial(s) that you want to fit (try a few!). You can even see the corresponding equations (and more) using the big arrow button →. If we select the "linear" and "5th degree polynomial" check boxes, and the "Show equations" check box, we will see the equations on the graph itself.


Basic Fitting - 1
Figure 1
File Edit View Insert Tools Desktop Window Help
Center and scale x data
Plot fits
Check to display fits on figure
☐ spline interpolant
☐ shape-preserving interpolant
☑ linear
☐ quadratic
☐ cubic
☐ 4th degree polynomial
☑ 5th degree polynomial
☐ 6th degree polynomial
☐ 7th degree polynomial
☐ 8th degree polynomial
☐ 9th degree polynomial
☐ 10th degree polynomial
☑ Show equations
☐ Plot residuals
Bar plot
Subplot
☐ Show norm of residuals
Help
Close
→
data 1
linear
5th degree
-2
0
2
4
6
8
10
12
14
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

Select data: data 1

Significant digits: 2

$y = - 2.5*x + 13$

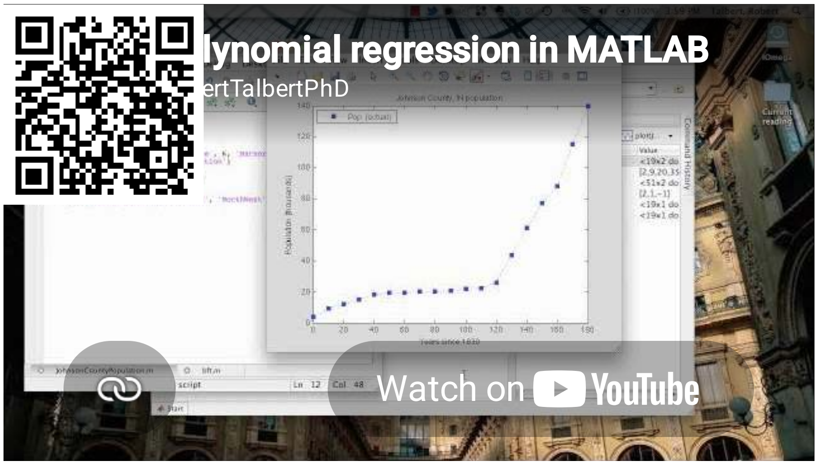
$y = - 0.017*x^5 + 0.33*x^4 - 2.1*x^3 + 4.7*x^2 - 4.9*x + 12$

This page titled 10.5: Interactive Fitting Tools is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Troy Siemers (APEX Calculus) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.3: Interactive Fitting Tools** by Troy Siemers is licensed CC BY-NC 3.0. Original source: https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

10.6: Chapter 10 Curve Fitting Videos in English and Espanol

10.6.1: English


lynomial regression in MATLAB
ertTalbertPhD
Watch on YouTube

.

atlab: Curve Fitting (polyfit and pol
l Holtschulte
rve Fitting
MATLAB
Watch on YouTube

.

10.6.2: Espanol

torial de MATLAB | Variables y Fu
er Maldonado
Watch on YouTube

.

gressão não linear com a rotina p
erTech
Watch on YouTube

.

This page titled 10.6: Chapter 10 Curve Fitting Videos in English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

11: Zero Finding and Fourier Transforms

11.1: Solving Nonlinear Equations with fzero()
11.2: fzero() Examples and Exercises
11.3: Using fzero() to find the intersection of 2 functions
11.4: Subfunctions
11.5: Anonymous (or In-Line) Functions
11.6: Debugging- More Name Collisions
11.7: Fourier Transforms
11.8: Chapter Review
11.9: Chapter 11 Zero finding Videos English and Espanol
11.10: Exercises

The next section describes how to solve for the roots (zeros) of a function called "error_func" using the fzero() function. The function is called "error_func" because the value it returns is the amount that the a particular value of x causes the function to differ from zero.

You can use fzero to solve for the zeros of a function with any name; your function does not need to be named "error_func".

This page titled 11: Zero Finding and Fourier Transforms is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

11.1: Solving Nonlinear Equations with fzero()

What does it mean to “solve” an equation? That may seem like an obvious question, but let’s take a minute to think about it, starting with a simple .

Suppose we want to know the value of a variable, x, but all we know about it is the relationship $x^2 = a$. If you’ve taken algebra, you probably know how to solve this equation: you take the square root of both sides and get $x = \pm\sqrt{a}$. Then, with the satisfaction of a job well done, you move on to the next problem.

But what have you really done? The relationship you derived is equivalent to the relationship you started with—they contain the same information about x—so why is the second one preferable to the first?

There are two reasons. One is that the relationship is now *explicit* in x: because x is all alone on the left side, we can treat the right side as a recipe for computing x, assuming that we know the value of a.

The other reason is that the recipe is written in terms of operations we know how to perform. Assuming that we know how to compute square roots, we can compute the value of x for any value of a.

When people talk about solving an equation, what they usually mean is something like “finding an equivalent relationship that is explicit in one of the variables.” In the context of this book, that’s what we’ll call an *analytic* , to distinguish it from a *numerical solution*, which is what we are going to do next.

To demonstrate a numerical solution, consider the equation $x^2 - 2x = 3$. You could solve this analytically, either by factoring it or by using the quadratic formula, and you would discover that there are two solutions, $x = 3$ and $x = -1$. Alternatively, you could solve it numerically by rewriting it as $x = \pm\sqrt{2x+3}$.

This equation is not explicit, since x appears on both sides, so it’s not clear that this move did any good at all. But suppose we had reason to expect a solution near 4. We could start with $x = 4$ as an *initial value* and then use the equation $x = \sqrt{2x+3}$ to compute successive approximations of the solution. (To understand why this works, see https://greenteapress.com/matlab/fixed.)

Here’s what happens:

```
>> x = 4;
>> x = sqrt(2*x+3)
x = 3.3166

>> x = sqrt(2*x+3)
x = 3.1037

>> x = sqrt(2*x+3)
x = 3.0344

>> x = sqrt(2*x+3)
x = 3.0114

>> x = sqrt(2*x+3)
x = 3.0038
```

After each iteration, `x` is closer to the correct answer, and after five iterations the relative error is about 0.1 percent, which is good enough for most purposes.

Techniques that generate numerical solutions are called *numerical method*. The nice thing about the method we just used is that it’s simple. But it doesn’t always work, and it’s not often used in practice. We’ll see a better alternative in the next section.

Zero-Finding

The MATLAB function `fzero` that uses numerical methods to search for solutions to nonlinear equations. In order to use it, we have to rewrite the equation as an *error function*, like this:

$$f(x) = x^2 - 2x - 3$$

The value of the error function is 0 if x is a solution and nonzero if it is not. This function is useful because we can use values of $f(x)$, evaluated at various values of x, to infer the location of the solutions. And that's what `fzero` does. Values of x where $f(x) = 0$ are called zeros of the function or *roots*.

To use `fzero` you have to define a MATLAB function that computes the error function, like this:

```
function res = error_func(x)
    res = x^2 - 2*x -3;
end
```

You can call `error_func` from the Command Window and confirm that 3 and -1 are zeros:

```
>> error_func(3)
ans = 0

>> error_func(-1)
ans = 0
```

But let's pretend that we don't know where the roots are; we only know that one of them is near 4. Then we could call `fzero` like this:

```
>> fzero(@error_func, 4)
ans = 3.0000
```

Success! We found one of the zeros.

The first argument is a *function handle* that specifies the error function. The `@` symbol allows us to name the function without calling it. The interesting thing here is that you're not actually calling `error_func` directly; you're just telling `fzero` where it is. In turn, `fzero` calls your error function—more than once, in fact.

The second argument is the initial value. If we provide a different value, we get a different root (at least sometimes).

```
>> fzero(@error_func, -2)
ans = -1
```

Alternatively, if you know two values that bracket the root, you can provide both.

```
>> fzero(@error_func, [2,4])
ans = 3
```

The second argument is a vector that contains two elements.

You might be curious to know how many times `fzero` calls your function, and where. If you modify `error_func` so that it displays the value of `x` when it is called and then run `fzero` again, you get

```
>> fzero(@error_func, [2,4])
x = 2
x = 4
x = 2.75000000000000
x = 3.03708133971292
x = 2.99755211623500
x = 2.99997750209270
x = 3.00000000025200
x = 3.00000000000000
x = 3
x = 3
ans = 3
```

Not surprisingly, it starts by computing $f(2)$ and $f(4)$. Then it computes a point in the interval, 2.75, and evaluates f there. After each iteration, the interval gets smaller and the guess gets closer to the true root. The `fzero` function stops when the interval is so small that the estimated zero is correct to about 15 digits.

If you'd like to know more about how `fzero` works, see Chapter 15.2.

What Could Go Wrong?

The most common problem people have with `fzero` is leaving out the `@`. In that case, you get something like so:

```
>> fzero(error_func, [2,4])
Not enough input arguments.

Error in error_func (line 2)
    res = x^2 - 2*x -3;
```

The error occurs because MATLAB treats the first argument as a function call, so it calls `error_func` with no arguments.

Another common problem is writing an error function that never assigns a value to the output variable. In general, functions should *always* assign a value to the output variable, but MATLAB doesn't enforce this rule, so it's easy to forget.

For example, if you write

```
function res = error_func(x)
    y = x^2 - 2*x -3
end
```

and then call it from the Command Window,

```
>> error_func(4)
y = 5
```

it looks like it worked, but don't be fooled. This function assigns a value to `y`, and it displays the result, but when the function ends, `y` disappears along with the function's workspace. If you try to use it with `fzero`, you get

```
>> fzero(@error_func, [2,4])
y = -3

Error using fzero (line 231)
FZERO cannot continue because user-supplied function_handle ==>
error_func failed with the error below.

Output argument "res" (and maybe others) not assigned during call
to "error_func".
```

If you read it carefully, this is a pretty good error message, provided you understand that “output argument” and “output variable” are the same thing.

You would have seen the same error message when calling `error_func` from the interpreter, if you had assigned the result to a variable:

```
>> x = error_func(4)
y = 5

Output argument "res" (and maybe others) not assigned during
call to "error_func".
```

Another thing can go wrong: if you provide an interval for the initial guess and it doesn’t actually contain a root, you get

```
>> fzero(@error_func, [0,1])
Error using fzero (line 272)
The function values at the interval endpoints must differ in sign.
```

There is one other thing that can go wrong when you use `fzero`, but this one is less likely to be your fault. It’s possible that `fzero` won’t be able to find a root.

Generally, `fzero` is robust, so you may never have a problem, but you should remember that there is no guarantee that `fzero` will work, especially if you provide a single value as an initial guess. Even if you provide an interval that brackets a root, things can still go wrong if the error function is discontinuous.

Choosing an Initial Value

The better your initial value is, the more likely it is that `fzero` will work, and the fewer iterations it will need.

When you’re solving problems in the real world, you’ll usually have some intuition about the answer. This intuition is often enough to provide a good initial guess.

If not, another way to choose an initial guess is to plot the error function and approximate the zeros visually. If you have a function like `error_func` that takes a scalar input variable and returns a scalar output variable, you can plot it with `ezplot`:

```
>> ezplot(@error_func, [-2,5])
```

The first argument is a function handle; the second is the interval you want to plot the function in. By examining the plot, you can estimate the locations of the two roots.

Vectorizing Functions

When you call `ezplot`, you might get the following warning (or error, if you’re using Octave):

```
Warning: Function failed to evaluate on array inputs;
vectorizing the function may speed up its evaluation and
avoid the need to loop over array elements.
```

This means that MATLAB tried to call `error_func` with a vector, and it failed. The problem is that it uses the `*` and `^` operators. With vectors, those operators don't do what we want, which is *element-wise* multiplication and exponentiation (see “Vector Arithmetic” on page 4.2).

If you rewrite `error_func` like this:

```
function res = error_func(x)
    res = x.^2 - 2.*x -3;
end
```

the warning message goes away, and `ezplot` runs faster.

This page titled 11.1: Solving Nonlinear Equations with fzero() is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **7.1: Solving Nonlinear Equations** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

11.2: fzero() Examples and Exercises

fzero() is a "function of a function", because it needs a "handle" to a function that defines the equation whose root it will find. fzero() can be used either to find a zero of a single functions and or to find the intersection point of 2 functions.

11.2.1: Using fzero() to find the root of a single function

The way it works is as follows: It finds an interval containing the initial point. It uses nearby points to approximate derivatives and estimate where the zero is. It then re-evaluates the function at this new x value. It repeats this procedure until it finds a point where the function is zero or very nearly zero.

✓ Example 11.2.1 Find a root of a 5th-order polynomial:

We will find a root of this 5th-order polynomial:

```
y = x.^5 - 4*x.^2 -10*x + 1;
```

Method 1: A straightforward way to do this is to create a function file for this:

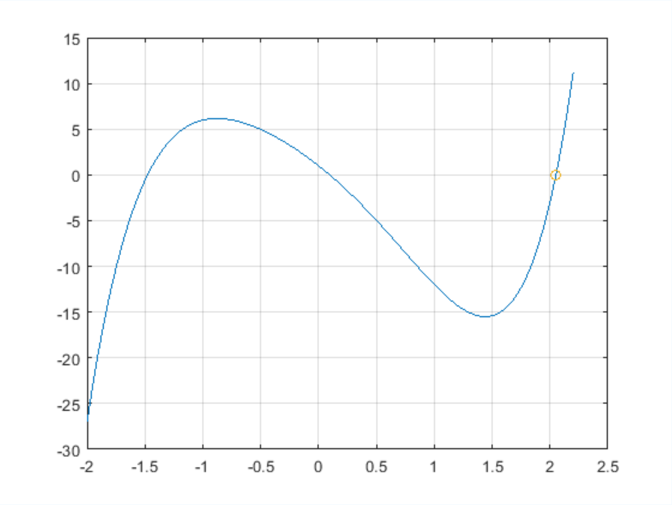
```
% fpoly5.m:

function y = fpoly5(x)

y = x.^5 - 4*x.^2 -10*x + 1;

end
```

Create a separate m-file script (fzero_poly5.m) to visualize this function and to estimate the root.

```
%% A. Open a new figure and plot the function
x = -2:0.05:2.2;

y = fpoly5(x);

figure

plot(x,y)

grid on;

%% B. the value of x near x=2.1 that makes y = 0

x_solution = fzero(@fpoly5, 2.1) % @fpoly5 is a function "handle" to the file fpoly5.m

% fzero() evaluates the function fpoly5(x) multiple times, until it converges to a root. (A root is a value that makes the function = 0)

% x_solution = 2.0517

% Verify that this is a solution:

yb = fpoly5(x_solution)

% 7.1054e-15

hold on;

plot(x_solution,yb,'o')

----
```

Method 2: Another way is to create a local sub-function at the bottom of the m-file script. This method currently works in MATLAB, but not Octave.

Solution



.

Exercise 11.2.1 fzero_y_fun1

```
%% A. Create this function
```

```
function y1 = y_fun1(x)
y1 = log(x)./x.^2 -0.1;
end
```

Create a separate m-file script to visualize this function and to estimate the root.

```
%% B. (1 pt) In this 2nd file, create an x vector from 0.2 to 2.0
% Choose an increment for x, such that the plot will be smooth--that is, no visible straight lines.
x = 1: 0.1 :6;
% (1 pt) Evaluate the equation using your x vector
y1 = y_fun1(x);
```

```
%% C. (1 pt) Open figure and plot this function.
% A plot of the function lets us estimate a root.
```

```
figure(1)
plot(x,y1)
grid on;
title('fzero1root\_fcnfile\_example.m CSmith')
hold on;
plot(x,zeros(size(x)),'r') % Plot a line for y = 0
%% D. (4 pts) Use the fzero() function to find a root near x = 4
```

Solution

```
x_solution = fzero(@y_fun1, 4)
% @y_fun1 = "function handle" to y_fun1.m
% The function y_fun1(x) must have a single input variable.
% fzero chooses the values of x for computing the function
% x_solution = 3.5656
```

%% E. fzero() can also be used to find a root between 2 values
x_solution = fzero(@y_fun1, [3,5])

.

Exercise 11.2.2 fzero for y = log10(x) + 0.44;

%% Create a function m-file that computes y = log10(x) + 0.44;
% Set
x = 0 : 0.02 : 1;

Create a separate m-file script to visualize this function and to estimate the root.

% Compute y for this x vector using your function.
% Open a figure and plot(x, y)
% Turn the grid on

% Use fzero() to find a root near 0.4 of this function

Answer

Add texts here. Do not delete this text first.

.

11.2.2:

This page titled 11.2: fzero() Examples and Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

11.3: Using fzero() to find the intersection of 2 functions

11.3.1: Using fzero() to find the intersection of 2 functions

Assume you have functions y1(x) and y2(x). Create a new function y3(x) = y1(x) - y2(x). The zero of y3(x) is the intersection of y1(x) and y2(x).

✓ Example 11.3.1 Find the intersection of cos(2x) and 3x

We will have 4 files. A test script file, and 3 function files: function files for each of the 2 functions and a function file that computes the difference of the 2 functions.

% A1. Create a file for a function that computes this straight line:

```
function y1 = fcn_line_03x(x)
    y1 = 0.3*x;
end
```

% A2. Create a file for a function that computes this cosine function:

```
function y2 = cos2x(x);
    y2 = cos(2*x);
end
```

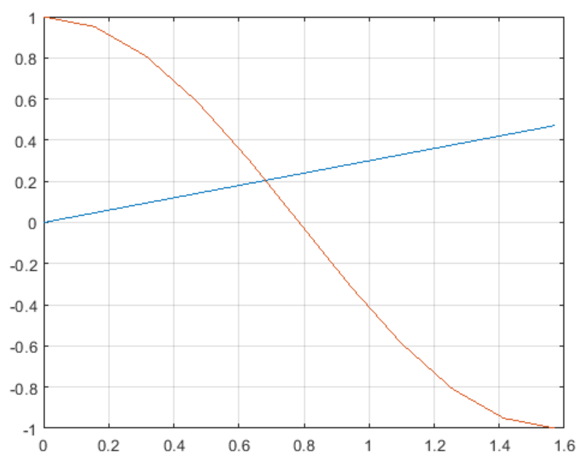
B. Create a function that computes the difference of these 2 functions:

```
function y3 = fcn_line_cos_diff(x)
    y3 = fcn_line_03x(x) - cos(2*x);
end
```

C. The test script m-file call the 2 functions in parts A1 and A2, opens a new figure, and plots the 2 functions:

```
x = (0: 0.02: 0.5)*pi;
y1 = fcn_line_03x(x);
y2 = cos(2*x);
figure(1)
plot(x,y1)
hold on;
plot(x,y2,'-s')
grid on;
axis equal
```

% We can see that one intersection of these functions is near x = 0.7 and y = 0.2



D. At the intersection point, y1(x) = y2(x). Rearranging, we get y1(x)- y2(x) = 0.

Create a function file that computes the difference between the 2 functions.

```
function y3 = fcn_line_cos_diff(x)
    y3 = fcn_line_03x(x) - cos(2*x);
end
```

So, were can use fzero() to find the point where the difference of the 2 functions is zero.

The code in the test script is:

```
x_solution = fzero(@fcn_line_cos_diff, 0.7) % near 0.7
```

% 0.6823

% Verify that this is a solution:

% Verify that this is a solution

```
y1b = fcn_line_03x(x_solution) % 1st function
```

% 0.2047

```
y2b = cos2x(x_solution) % 2nd function
```

% 0.2047

%% Plot the solution point on the 1st figure

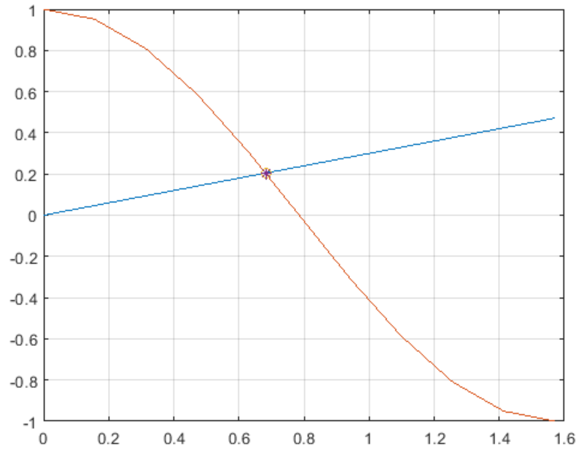
```
figure(1)
hold on;

plot(x_solution,y1b,'bo','Markersize', 10)
plot(x_solution,y2b,'c*','Markersize', 10)
```

? Exercise 11.3.1

Add exercises text here.

Answer

Add texts here. Do not delete this text first.



Solution

Add example text here.

.

Exercise 11.3.1 Intersection of quadratic and cubic functions

A. (2 pts) Create these 2 function files:

```
y_fun_x2(x)
   y_x2 = x.^2 -4;
end
function y_x3 = y_fun_x3(x)
   y_x3 = x.^3 - 6*x;
end
```

B1. (2 pt) Begin a script m-file named `fzero_y_fun_x2_x3_YourName.m` that will plot these functions and will solve for one of their intersections.

At the top of this script, include this code:

```
clear all; clear functions; close all; format compact; clc;
```

B2a. (1 pt) Create an x vector from -3 to 3
Choose an increment for x = 0.3, so that the plot will be smooth.
B2b. (2 pts) Use this x vector to compute the corresponding values of y for each of the 2 functions.

```
y2 = y_fun_x2(x);
y3 = y_fun_x3(x);
```

B3. (2 pts) Open a figure and plot y2, add "hold on;", the plot y3 on the figure. You will see that they intersect at 3 places.

C1. (2 pts) fzero() finds a root of a single function, so create a 3rd function that computes the difference of the 1st 2 functions:

```
function y23 = y_2_3_fun(x)
   y23 = y_fun_x2(x) - y_fun_x3(x);
end
```

Use the x vector to compute values of y for the difference function:

```
y23 = y_2_3_fun(x)
```

C2. (1 pt) Open a 2nd figure and plot(x, y23)

D. (6 pts) Use the fzero() function to find a solution where y_2_3_fun(x) = 0.

Use the function handle, `@y_2_3_fun` as the 1st input of fzero().

For the initial value of x, choose an integer value (whole number) near any one of the intersection points. (Do not use a decimal value that is not an integer.)

Answer

Add texts here. Do not delete this text first.

.

? Exercise 11.3.2 fzero_exp

A. Set

x = (0 : 0.05 : 1)

B. Compute these 2 functions:

y1 = 0.7x

y2 = exp(-2*x)

C. Open a new figure and plot these 2 functions on it.

plot(x,y1)

hold on;

plot(x,y2)

The intersection of these 2 functions is the solution of this transcendental equation:

y1 = y2 or, equivalently, y1 - y2 = 0

D. Find the value of x near 0.5 that is a solution of this equation using the fzero() function as follows:

D1. Rewrite it in the form: fexp(x) = exp(-2*x) - 0.7*x, then create a function file called fexp.m that computes this.

D2. Solve it using the fzero() function, using @fexp as the function handle.

Answer

Add texts here. Do not delete this text first.

.

This page titled 11.3: Using fzero() to find the intersection of 2 functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

11.4: Subfunctions

There is an option in MATLAB to put a function at the bottom of a script or function file. Such a function is called a subfunction.

Subfunctions are never necessary, but can be convenient if the function is a particular function for only 1 script. A general purpose user function should be put in its own file.

In Octave, a subfunction can only be put in a function file, not in a script file.

Consider the exercise from the previous section with the script fzero_y_fun1.m and the function y_fun1.m

The function could be put at the bottom of the script shown in this example:

✓ Example 11.4.1 y_fun1 subfunction

```
x = 1: 0.1 :6;
% (1 pt) Evaluate the equation using your x vector
y1 = y_fun1(x);

%% C. (1 pt) Open figure and plot this function.
% A plot of the function lets us estimate a root.
figure(1)
plot(x,y1)
grid on;
title('fzero1root\_fcnfile\_example.m CSmith')
hold on;
plot(x,zeros(size(x)),'r') % Plot a line for y = 0
%% D. (4 pts) Use the fzero() function to find a root near x = 4
x_solution = fzero(@y_fun1, 4)
% x_solution = 3.5656

%% E. fzero() can also be used to find a root between 2 values
x_solution = fzero(@y_fun1, [3,5])

%%
function y1 = y_fun1(x)
y1 = log(x)./x.^2 -0.1;
end
```

Solution

Add example text here.

This page titled 11.4: Subfunctions is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

11.5: Anonymous (or In-Line) Functions

There is an option in MATLAB to create an in-line function, without creating a regular function file or subfunction. It only works if the function is simple enpough to be written on a single line of code. The syntax of this option is confusing to most students. This option is never necessary, but can be convenient if the function is a particular function for only 1 script. Such in-line functions are called often "anonymous" functions. This name "anonymous" functions is common, but misleading, since such a function does have a name. (Anonymous functions are an advanced topic and is optional for this textbook.) A general purpose user function should be put in its own file.

Using the example of the previous section, the function can be defined as an in-line function with this syntax:

```
y_fun1 = @(x) log(x)./x.^2 -0.1;
```

The left side states the name of the function.

The @(x) states that the function is a function of x.

The rest of the right side is the formula that will be computed.

For an in-line (anonymous) function the function name is the function handle, so when we use it in fzero, we use the name without the @ as shown here:

```
x_solution = fzero(y_fun1, 4)
```

.

This page titled 11.5: Anonymous (or In-Line) Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

- **6.6: Anonymous (or In-Line) Functions** by Masayuki Yano, James Douglass Penn, George Konidaris, & Anthony T Patera is licensed CC BY-NC-SA 4.0. Original source: https://ocw.mit.edu/courses/2-086-numerical-computation-for-mechanical-engineers-spring-2013.

11.6: Debugging- More Name Collisions

When you start writing longer programs, you might spend more time debugging. So we'll end this chapter with some debugging tips.

More Name Collisions

Functions and variables occupy the same workspace, which means that whenever a name appears in an expression, MATLAB starts by looking for a variable with that name; if there isn't one, it looks for a function.

As a result, if you have a variable with the same name as a function, the variable *shadows* the function; metaphorically, you can't see the function because the variable is in the way.

For example, if you assign a value to `sin` and then try to use the `sin` function, you might get an error:

```
>> sin = 3;
>> sin(5)
Index exceeds the number of array elements (1).

'sin' appears to be both a function and a variable.
If this is unintentional, use 'clear sin' to remove
the variable 'sin' from the workspace.
```

Since the value we assigned to `sin` is a number, and a number is considered a 1×1 matrix, MATLAB tries to access the fifth element of the matrix and finds that there isn't one.

In this case, MATLAB is able to detect the error, and the error message is pretty helpful. But if the value of `sin` were a vector, or if the argument were smaller, you would be in trouble. For example,

```
>> sin = 3;
>> sin(1)
ans = 3
```

Just to review, the sine of 1 is not 3!

You can avoid these problems by choosing function names carefully. Use long, descriptive names for functions, not single letters like `f`. To be even clearer, use function names that end in `func`. And before you define a function, check whether MATLAB already has a function with the same name.

Debugging Your Head

When you're working with a new function or a new language feature for the first time, you should test it in isolation before you put it into your program.

For example, suppose you know that `x` is the sine of some angle and you want to find the angle. You find the MATLAB function `asin`, and you're pretty sure it computes the inverse sine function. Pretty sure is not good enough; you want to be very sure.

Since we know $\sin 0 = 0$, we could try

```
>> asin(0)
ans = 0
```

which is correct. Now, we also know that the sine of 90° is 1, so if we try `asin(1)`, we expect the answer to be 90, right?

```
>> asin(1)
ans = 1.5708
```

Oops. We forgot that the trig functions in MATLAB work in radians, not degrees. The answer we got is $\pi/2$, which is 90°, in radians.

With this kind of testing, you’re not really checking for errors in your program; you’re checking your understanding. If you make an error because you are confused about how MATLAB works, it might take a long time to find, because when you look at the code, it looks right.

Which brings us to the Seventh Theorem of Debugging:

> *The worst bugs aren’t in your code; they’re in your head.*

This page titled 11.6: Debugging- More Name Collisions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **7.2: Debugging** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

11.7: Fourier Transforms

By Carey A. Smith

The code in this section is in file: FFT_demo_1.m

FFT_demo_1.m

The Fourier Transform is defined as the integral of a signal multiplied by sine and cosine functions of frequencies which are multiples of a fundamental frequency.

A common use of FFT's is to find the frequency components of a signal.

% Computers compute the Fourier Transform of a signal vector using an algorithm called the "Fast Fourier Transform" or FFT.

MATLAB's function is fft(). The input is a vector y of signal values sampled at times specified by a vector t. For a signal vector, y, and a time vector, t, the syntax to properly scale the frequencies is:

```
y_FFT = (sqrt(t(end)))*fft(y);
```

Example 11.7.1 Spectral Analysis 1

```
%% This example shows the use of the Fourier Transform for spectral analysis.
clear all; close all; format compact; format short; clc;
% Create a time vector
nt = 256; % Vector length (A power of 2 is most efficient)
dt = 0.01; % (sec) Sampling interval
t = (0:1:(nt-1))*dt; % (sec) Time vector
%% Create a Signal vector
bias = 0; % Average value
f1 = 4.0 % (Hz) 1st frequency
a1 = 4.0 % Magnitude of the 1st frequency
f2 = 10.0 % (Hz) 2nd frequency
a2 = 2.0 % Magnitude of the 2nd frequency
y = bias + a1*sin(2*pi*f1*t) + a2*sin(2*pi*f2*t);
y_power = mean(y.^2) % power in the signal
%% Plot y vs. t
figure;
plot(t,y)
title('Original Signal vs. Time')
xlabel('Time (s)')
grid on;
```

```
%% Compute the Fourier Transform

y_FFT = (sqrt(t(end)))*fft(y); % Scale the FFT

f_max = (1/dt) % (Hz) Maximum frequency = Sample frequency

df = (1/dt)/nt % (Hz) Freq. interval

freqs = (0:1:(nt-1))*df; % Frequencies of the FFT

%% Plot the FFT of y

% The FFT result vector has complex values

figure;

plot(freqs,real(y_FFT),'b');

hold on;

plot(freqs,imag(y_FFT),'r');

legend('Real(y\_FFT)','Imag(y\_FFT)');

title('FFT(y))');

xlabel('Frequency (Hz)');

grid on;

%% Convert the complex Fourier Transform to magnitude squared.

y_FFT2 = (df/nt^2)*(y_FFT.*conj(y_FFT)); % (df/nt^2)is a scaled factor

y_FFT2_power = sum(y_FFT2)

% 10.02

figure;

plot(freqs,y_FFT2);

title('FFT2 Power)');

xlabel('Frequency (Hz)');

grid on;

%% The Nyquist–Shannon sampling theorem says

% you can only measure frequencies up to 1/2 the sampling frequency.

f_Nyquist = f_max/2

% You can see that the power plot is symmetric about the center frequency.

% The frequencies > f_Nyquist are equivalent to negative frequncies.

% We double the lower frequncies (except for 0 Hz)

% and delete the high frequencies.
```

```
y_FFT3 = [y_FFT2(1), 2*y_FFT2(2:(nt/2)), y_FFT2(nt/2+1)];

y_FFT3_power = sum(y_FFT3)

% 10.02

freqs3 = freqs(1:(nt/2+1));

figure;

plot(freqs3,y_FFT3);

title('y\_FFT3 Power Spectrum');

xlabel('Frequency (Hz)');

grid on;
```

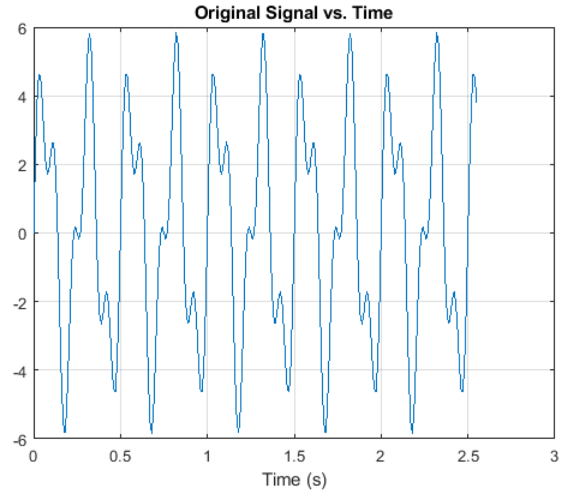


Figure 11.7.1: Signal vs. time

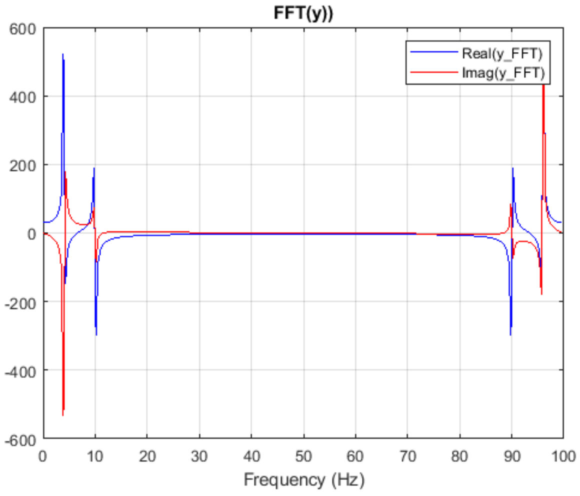


Figure 11.7.1: The FFT's real and imaginary components

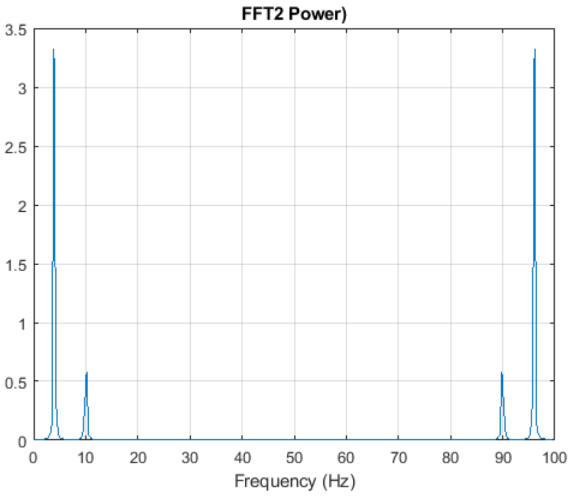


Figure 11.7.1: Power Spectrum, 0 to 100 Hz

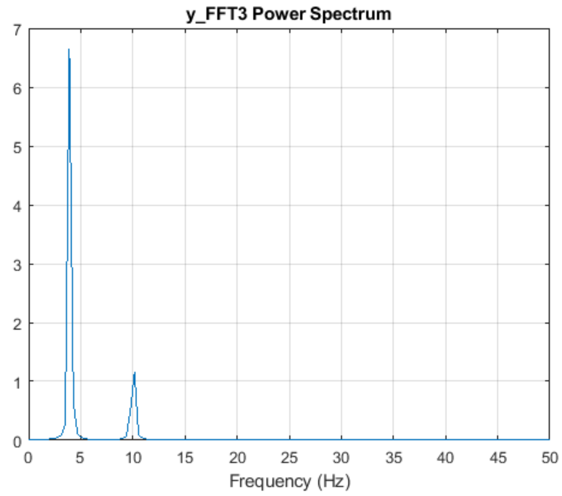


Figure 11.7.1: Power Spectrum, folder onto 0 to 50 Hz

.

? Exercise 11.7.1 Spectral Analysis 2

Modify the code in the above example as follows.

Change the 2 frequencies to 6 Hz and 16 Hz.

Call this signal y0. (This is without noise)

Use randn() to add a noise vector to y0 with a mean of 0 and a standard deviation of 4.

y = y0 + 4*randn(size(y));

This adds noise of all frequencies. Such noise is called “white noise”.

The Power Spectrum should be strong at the 2 main frequencies. A small amount of random power should be at all the other frequencies.

Answer

Add texts here. Do not delete this text first.

? Exercise 11.7.2 Spectral Analysis with large noise

Using your code from the previous exercise (Spectral Analysis 2), increase the noise to a point where the noise makes it somewhat hard to see the periodic part of signal in the plot of y vs. t.

Describe what the Fourier Transform plot shows.

Are the frequencies of the 2 sine waves identifiable?

Answer

Add texts here. Do not delete this text first.

This page titled 11.7: Fourier Transforms is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

11.8: Chapter Review

This chapter introduced `fzero`, a function we can use to solve nonlinear equations. To use `fzero`, you have to write an error function and pass a function handle as an argument. Using functions in this way can be tricky at first, but get comfortable with it, because we are going to use it a lot.

Here are some terms from this chapter you might want to remember.

If we can solve an equation by performing algebraic operations and deriving an explicit way to compute a value, the result is an *analytic solution*. Otherwise, we can use a *numerical method*, which finds a *numerical solution* to the equation, which is usually an approximation.

To solve nonlinear equations, we often rewrite them as functions and then find one or more *zeros* of the function, that is, arguments that make the value of the function 0.

A *function handle* is a way of referring to a function by name (and passing it as an argument) without calling it.

Finally, *shadowing* is a kind of name collision in which a new definition causes an existing definition to become invisible. In MATLAB, variable names can shadow built-in functions (with hilarious results).

In the next chapter, we'll write functions that take vectors as inputs and return vectors as outputs.

This page titled 11.8: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

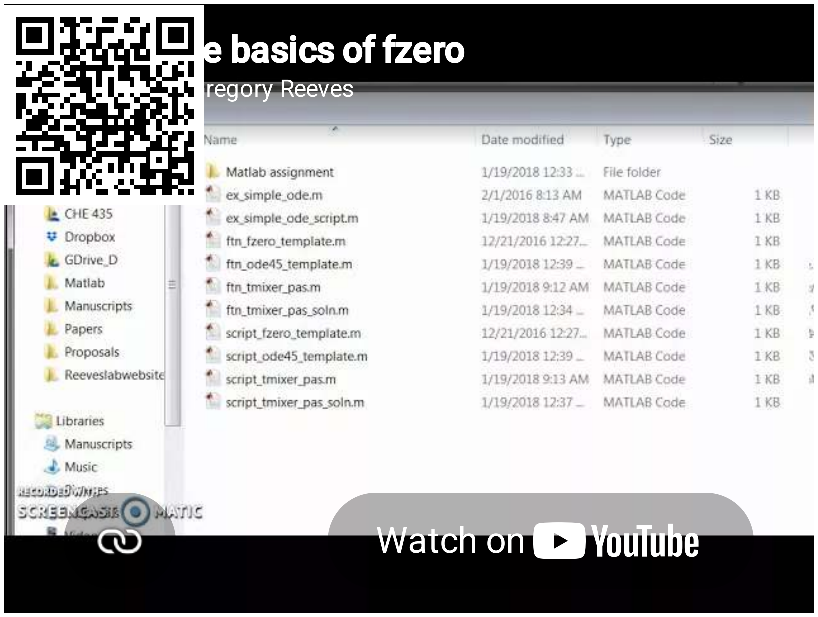
- **7.3: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

11.9: Chapter 11 Zero finding Videos English and Espanol

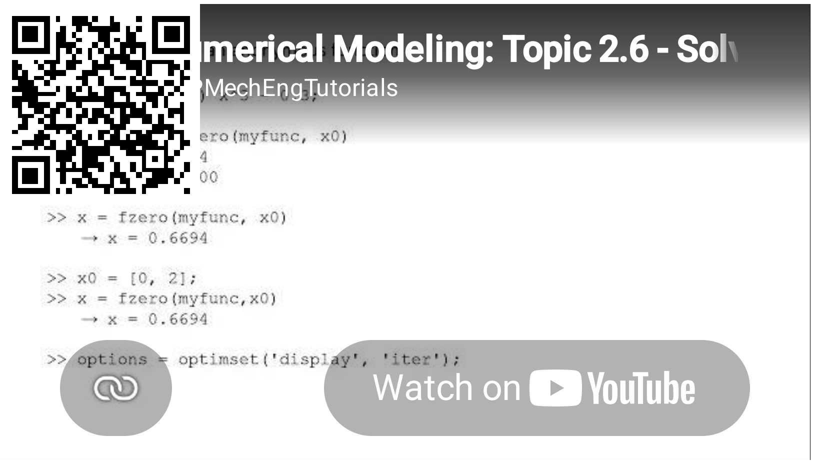
English

The basics of fzero


e basics of fzero
regory Reeves
CHE 435
Dropbox
GDrive_D
Matlab
Manuscripts
Papers
Proposals
Reeveslabwebsite
Libraries
Manuscripts
Music
RECORDED WITH SCREENCAST-O-MATIC
Watch on YouTube

Name	Date modified	Type	Size
Matlab assignment	1/19/2018 12:33 ...	File folder	
ex_simple_ode.m	2/1/2016 8:13 AM	MATLAB Code	1 KB
ex_simple_ode_script.m	1/19/2018 8:47 AM	MATLAB Code	1 KB
ftn_fzero_template.m	12/21/2016 12:27...	MATLAB Code	1 KB
ftn_ode45_template.m	1/19/2018 12:39 ...	MATLAB Code	1 KB
ftn_tmixer_pas.m	1/19/2018 9:12 AM	MATLAB Code	1 KB
ftn_tmixer_pas_soln.m	1/19/2018 12:34 ...	MATLAB Code	1 KB
script_fzero_template.m	12/21/2016 12:27...	MATLAB Code	1 KB
script_ode45_template.m	1/19/2018 12:39 ...	MATLAB Code	1 KB
script_tmixer_pas.m	1/19/2018 9:13 AM	MATLAB Code	1 KB
script_tmixer_pas_soln.m	1/19/2018 12:37 ...	MATLAB Code	1 KB

MATLAB's fzero function video


merical Modeling: Topic 2.6 - Sol
MechEngTutorials
Watch on YouTube

```
ero(myfunc, x0)
4
00
>> x = fzero(myfunc, x0)
   → x = 0.6694

>> x0 = [0, 2];
>> x = fzero(myfunc,x0)
   → x = 0.6694

>> options = optimset('display', 'iter');
```

Espanol

NO LINEALES en Matlab 💜 [fsolve, fzero, roots]

stema de Ecuaciones NO LINEALE
io A. Castaño Giraldo
O Lineales
fsolve
fzero
roots
44
Watch on YouTube

This page titled 11.9: Chapter 11 Zero finding Videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

11.10: Exercises

Before you go on, you might want to work on the following exercises.

Exercise 7.1

1. Write a function called `cheby6` that evaluates the sixth Chebyshev polynomial. It should take an input variable, x, and return

$$32x^6 - 48x^4 + 18x^2 - 1$$

2. Use `ezplot` to display a graph of this function in the interval from -1 to 1. Estimate the location of any zeros in this range.
3. Use `fzero` to find as many different roots as you can. Does `fzero` always find the root that is closest to the initial value?

Exercise 7.2

When a duck is floating on water, how much of its body is submerged?[1]

To estimate a solution to this problem, we'll assume that the submerged part of a duck is well approximated by a section of a sphere. If a sphere with radius r is submerged in water to a depth d, the volume of the sphere below the water line is

$$V = \frac{\pi}{3}(3rd^2 - d^3) \quad \text{as long as} \quad d < 2r$$

We'll also assume that the density of a duck is $\rho = 0.3\,\text{g/cm}^3$ (0.3 times the density of water) and that its mass is $\frac{4}{3}\pi r^3 \rho\,\text{g}$.

Finally, according to the law of buoyancy, an object floats at the level where the weight of the displaced water equals the total weight of the object.

Here are some suggestions for how to proceed:

1. Write an equation relating ρ, d, and r.
2. Rearrange the equation so the right-hand side is zero. Our goal is to find values of d that are roots of this equation.
3. Write a MATLAB function that evaluates this function. Test it, then make it a quiet function.
4. Make a guess about the value of d_0 to use as an initial value.
5. Use `fzero` to find a root near d_0.
6. Check to make sure the result makes sense. In particular, check that $d < 2r$, because otherwise the volume equation doesn't work!
7. Try different values of ρ and r and see if you get the effect you expect. What happens as ρ increases? Goes to infinity? Goes to zero? What happens as r increases? Goes to infinity? Goes to zero?

---

[1] This exercise is adapted from C. F. Gerald and P. O. Wheatley, *Applied Numerical Analysis*, 4th edition (Boston: Addison-Wesley, 1989).

This page titled 11.10: Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **7.4: Exercises** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

CHAPTER OVERVIEW

12: MATLAB Matrices (Arrays)

12.1: Creating Matrices and Arrays
12.3: Common Matrix Functions
12.4: Matrix Indexing
12.5: Matrix Operations, Introduction
12.6: Matrix Tables and Nested Loops
12.7: Dot Product
12.8: Cross Product and Torque
12.9: Chapter 12 Matrix Videos English and Espanol

This page titled 12: MATLAB Matrices (Arrays) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

12.1: Creating Matrices and Arrays

By Carey A. Smith

For our purposes, an array is 2-dimensional set of numbers.

A matrix is a 2-dimensional set of numbers that is used for linear algebra computations.

MATLAB keeps track of the numbers of rows and columns of a matrix.

12.1.1: User-defined matrices are typically defined in the following 2 equivalent ways:

Method 1: Create a matrix in directly 2-dimensions

```
M1 = [1 4 7 10
      2 5 8 11
      3 6 9 12]
```

Method 2: Create a matrix on 1 line, with semicolons separating the rows:

```
M2 = [1 4 7 10; 2 5 8 11; 3 6 9 12]
```

This creates the same matrices, but is not as visually obvious. This method is often used in books to save space on a page.

It is also possible to define MATLAB matrices (arrays) with 3 or more dimensions, but we will only use 2-dimensional arrays in this text.

12.1.2: Built-in functions to create a matrix

zeros(rows,cols)

Example:

```
z23 = zeros(2,3)
```

Result:
0 0 0
0 0 0

ones(rows,cols)
Example:

```
a32 = ones(3,2)
1 1
1 1
1 1
```

You can create a matrix of all constant values by multiplying a "ones" matrix by a constant.

6*ones(rows,cols)

```
a32_6 = 6*ones(3,2)
6 6
6 6
6 6
```

eye(n) = Identity matrix
Example:

```
I4 = eye(4)
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

magic(n) = Magic square. The sums of all the rows, columns, and diagonals are the same in a magic square.

Example:

```
m3 = magic(3)
8 1 6
3 5 7
4 9 2
```

diag(v) When v is a vector, creates a square matrix with diagonal elements = v.

```
v = [4, 9, 16]
B = diag(v)
4 0 0
0 9 0
0 0 16
```

diag(M) When M is a matrix, this extracts the diagonal elements of matrix M.

Example using magic square m3:

```
diag(m3) = [8
            5
            2]
```

```
pascal4 = pascal(4)
1  1  1  1
1  2  3  4
1  3  6 10
1  4 10 20
```

It has binomial coefficients

12.1.3: Matrix Transpose

Previously, we saw that a row vector can be transposed to change it to a column vector, like this:

```
a = [1 2 3]
```

```
at1 = a'
```

This becomes:

```
at1 =
     1
     2
     3
```

You can also use the transpose( ) function:

```
at2 = transpose(a)
```

```
at2 =
     1
     2
     3
```

The same function can be applied to any matrix, as shown by these examples:

Set

```
m3 = magic(3)
```

```
m3 = 8 1 6
     3 5 7
     4 9 2
```

```
m3t = m3'
```

```
m3t =
     8     3     4
     1     5     9
     6     7     2
```

Set

```
m24 = [1 2

       3 4

       5 6


       7 8]
```

```
m24t = m24'
```

```
m24t =
     1     3     5     7
     2     4     6     8
```

This page titled 12.1: Creating Matrices and Arrays is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

12.3: Common Matrix Functions

Commands: size and length

Many functions involving matrices will only work if the dimensions of the matrices satisfy certain conditions. The size(M) command returns the number of rows and columns in `M`.

✓ Example 11.6.1

```
>> A = [1 2 3 4;
        5 6 7 8];
>> size(A)
ans =
2 4
```

and

```
>> B = [1;
        6;
        0;
        9];
>> size(B)
ans =
4 1
```

The `size()` command is also what is known as an “overloaded” function in that it can be used in a couple of ways. Suppose we only wanted to know only the number of rows in a matrix M. We could find the `size(M)`, store this as a variable and then select the first entry. Instead, the `size()` command takes a second entry that will allow us to get what we want.

✓ Example 11.6.2

```
>> A = [1 2 3 4;
        5 6 7 8];
>> size(A,1)
ans =
2
```

(the number of rows)

and

```
>> A = [1 2 3 4;
        5 6 7 8];
>> size(A,2)
ans =
4
```

(the number of columns).

The length of a vector (either a row or column) is simply the number of elements in the vector:

Example 11.6.3

```
>> c=1:5;

>> length(c)

ans =
5
```

However, the “length of a matrix” is defined in Matlab as the larger of the number of rows and the number of columns. That is, length(A) is equivalent to max(size(A)).

Commands: max and min

These two functions are (almost) self explanatory. For the matrix that we are using,

```
A = [1 2 3 4;

     5 6 7 8],
```

if we try `max(A)`, we get `5 6 7 8`. What’s going on?

Remember that Matlab works on column precedence so that what max() is doing is not finding the maximum value of the entire matrix, but instead finding the maximums of each column. The only exception occurs when the starting matrix is either a row or column vector. For example, for `B = [1; 6; 0; 9]`, `max(B)` does give us `9` (the largest value in `B`).

So, to get the largest element in `A`, we could “nest” the functions as `max(max(A))`, which would give us `8`.

Or, we can use max() with the colon operator to treat the entire matrix as a vector: max(A(:)), which would also compute `8`. The form max(A(:)) is preferred, because this form works with other functions such as std(A(:)) which give the correct standard deviation, while std(std(A)) gives an incorrect answer. [C.A.S.]

Commands: sum and prod

Once again, these seem reasonably named functions (see previous bullet). And once again, they return not the sum\product of every entry in matrix, but the column sums\products with the exception being for vectors, in which case you do get the sum\product of every entry in the vector. But, what if you want to get the sum along the rows? Well, once again sum is an overloaded operator and we can use:

Example 11.6.4

```
>> A = [1 2 3 4;

        5 6 7 8]

>> sum(A)

ans =
6 8 10 12
```

(the column sums

and

```
>> sum(A,2)% The "2" means sum in the 2nd dimension, that is, across the rows

ans =
10
26
```

(the row sums)

For either a row vector, V, or a column vector, V, sum(V) is the syum of the elements of V.

For a matrix, A, we have the following options when using sum():

sum(A) sums the columns (the "1st dimension")

sum(A, dim): dim means the direction of the sum:

When A is 2-dimensional:

sum(A, 1) is the sum of the columns.

sum(A, 2) is the sum of the rows.

To create a 3-dimensional matrix, read this link:

https://www.mathworks.com/help/matlab/math/multidimensional-arrays.html

.

Exercise

Exercise 12.3.1 "Hilbert matrix" operations

Consider the 5 x 5 "Hilbert matrix"

```
H5 = [ 1  1/2 1/3 1/4 1/5;
      1/2 1/3 1/4 1/5 1/6;
      1/3 1/4 1/5 1/6 1/7;
      1/4 1/5 1/6 1/7 1/8;
      1/5 1/6 1/7 1/8 1/9];
```

a. Using a command in this section, find the dimensions of H5.
b. Using commands in this section, find the column sums of H5 and the row sums of H5.
c. Use the max() function to determine the value of the maximum entry of H5.

Answer

Add texts here. Do not delete this text first.

.

This page titled 12.3: Common Matrix Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Troy Siemers (APEX Calculus) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.4: Common Matrix Functions** by Troy Siemers is licensed CC BY-NC 3.0. Original source: https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

12.4: Matrix Indexing

Let’s look at how we can reference parts of a matrix.

✓ Example 11.3.1

Consider the matrix A = [1 2 3 4;5 6 7 8]. There are actually two ways to view this matrix, either as a rectangular array of 2 rows and 4 columns, or as a list of 8 elements. Suppose we wanted to isolate the 7 in the matrix A and store it as the variable temp. First, we can think of the 7 as being located in the second row and third column. In this case, we can type:

```
>> A = [1 2 3 4;5 6 7 8];
>> temp = A(2,3)
```

with the result being:

```
temp =
7
```

Second, we can think of 7 as being one of the eight elements total. But, it is crucial to realize that we count elements in this way using a “column precedence.” This means that we count, one at a time, down the columns. This means that we can think of 7 as being located in the 6th entry, or:

```
>> temp = A(6)
```

also gives the result:

```
temp =
7
```

For completeness, in the last example, if we think of A as a matrix:

```
A(1,1) = 1 A(1,2) = 2 A(1,3) = 3 A(1,4) = 4
A(2,1) = 5 A(2,2) = 6 A(2,3) = 7 A(2,4) = 8
```

or, thinking of A as a vector:

```
A(1) = 1 A(3) = 2 A(5) = 3 A(7) = 4
A(2) = 5 A(4) = 6 A(6) = 7 A(8) = 8
```

If we wanted to store the entire first row of A in the variable `firstrow,` we would say that we want “all four columns of the first row.” This suggests that we can use the colon operator to shorten our work. Namely,

```
>> firstrow = A(1,1:4)
```

which gives

```
firstrow =
1 2 3 4
```

But, there’s an even shorter way to do this! If the colon doesn’t have a start and end value, it simply lists all possible values! Namely,

```
>> firstrow = A(1,:)
```

also gives

```
firstrow =
1 2 3 4
```

Ok, now what if we wanted the first row, but not the element in the first column? There are two ways to do this. First, we can use the colon as:

```
>> mostoffirstrow = A(1,2:4)
```

which gives

```
mostoffirstrow =
2 3 4
```

But, what if the matrix changes and we don't know how big A has changed to? Those sneaky programmers at Mathworks have a work around:

```
>> mostoffirstrow = A(1,2:end)
```

also gives

```
mostoffirstrow =
2 3 4
```

? Exercise 12.4.1 "Hilbert matrix" rows

Consider the 5 x 5
H5 = [1 1/2 1/3 1/4 1/5;
1/2 1/3 1/4 1/5 1/6;
1/3 1/4 1/5 1/6 1/7;
1/4 1/5 1/6 1/7 1/8;
1/5 1/6 1/7 1/8 1/9];

By referencing H5, create a matrix that consists of the 2nd and 3rd rows of H5 (Do not simply retype the entries).

Answer

Add texts here. Do not delete this text first.

.

This page titled 12.4: Matrix Indexing is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Troy Siemers (APEX Calculus) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.2: Using Matrices** by Troy Siemers is licensed CC BY-NC 3.0. Original source: https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

12.5: Matrix Operations, Introduction

Here we will explore the algebra of matrices. It would be wise for the reader to have a basic knowledge of matrix algebra (or even linear algebra), but we will
try and give plenty of explanatory examples.

Just as for scalars, many of the common algebraic operations apply to matrices. The symbols + and − carry over quite nicely in “element-by-element” operations as one would expect (or at least hope for). Similarly, if you want element-by-element multiplication, division, or even exponentiation, these are given by `.∗`, ./ and `.∧` (yes, those each have a preceding period and are pronounced **“dot times”**, **“dot divide”** and **“dot exponent”**). Operations like “regular” multiplication, division, exponents, etc. have a very different meaning than one who hasn’t been exposed to linear algebra might expect. We also have operations like the matrix transpose.

Let’s look at some examples using the matrices `A = [1 2 3 4;5 6 7 8]` and `B = [1; 6; 0; 9].` Again,

```
A =
1 2 3 4
5 6 7 8
```

```
and
B =
1
6
0
9
```

Example 11.4.1: Scalar Multiplication

Here we show how to multiply every entry in a matrix by a scalar:

```
>> A = [1 2 3 4;5 6 7 8];
```

```
>> 3*A
```

gives

```
ans =
3 6 9 12
15 18 21 24
```

One special operation for matrices is the matrix **transpose**, given by either transpose(A) or the shorter A'. The transpose of a matrix is another matrix with the rows and columns interchanged.

Example 11.4.2: Transpose

```
>> A = [1 2 3 4;5 6 7 8];
```

```
>> A'
```

```
ans =
1 5
2 6
3 7
4 8
```

and

```
>> B = [1; 6; 0; 9];B'
```

```
ans =
1 6 0 9
```

The example `A∧2` gives an error saying A must be square (here it is good to know some linear algebra), but `A.∧2` gives element-by-element squaring:

✓ Example 11.4.3: Element by element squaring

```
>> A = [1 2 3 4;5 6 7 8];

>> A.∧2

ans =

1 4 9 16
25 36 49 64
```

Finally, we can create “block” matrices from smaller matrices by treating them as elements themselves and using the comma (or space) and the semicolon to create rows and columns. We just have to make sure the dimensions of the matrices line up properly. For example, we can stack two matrices.

✓ Example 11.4.4: Stacking Matrices

```
>> A = [1 2 3 4;5 6 7 8];

>> B = [1; 6; 0; 9];

>> AoverBprime = [A ; B']

AoverBprime =
1 2 3 4
5 6 7 8
1 6 0 9
```

This page titled 12.5: Matrix Operations, Introduction is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Troy Siemers (APEX Calculus) via source content that was edited to the style and standards of the LibreTexts platform.

- **2.3: Matrix Operations** by Troy Siemers is licensed CC BY-NC 3.0. Original source: https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

12.6: Matrix Tables and Nested Loops

By Carey A. Smith

A table with 2 or more related rows or columns can be created as a matrix, as shown in these examples.

Example 12.6.1 Row matrix table of squares and cubes

```
x1 = 13:16
% x1 = 13 14 15 16
x2 = x1.^2 % square each element of x1
% x2 = 169 196 225 256
x3 = x1.^3 % cube each element of x1
% x3 = 2197 2744 3375 4096
% These can be put into a single variable that is a horizontal table relating these values:
x_table = [x1
x2
x3]
% x_table =
% 13 14 15 16
% 169 196 225 256
% 2197 2744 3375 4096
```

Solution

Example 12.6.2 Column matrix table of squares and cubes

```
%% Often,a vertical table is preferred, especially if there are a lot of values.
y1 = 11:20;
y2 = y1.^2; % square each element of y1
y3 = y1.^3; % cube each element of y1
% A vertical table is created by transposing each horizontal vector into a vertical vector:
y_table = [y1' y2' y3']
% y_table =
% 11 121 1331
% 12 144 1728
% 13 169 2197
% 14 196 2744
% 15 225 3375
% 16 256 4096
% 17 289 4913
% 18 324 5832
% 19 361 6859
% 20 400 8000
```

Solution

Add example text here.

✓ Example 12.6.3 km to miles matrix table

```
%% Example of km to miles
% In this case, each vector is created as a vertical vector, so the transpose is not needed when creating the table:
km = (1:10)' % The ' is the transpose operator.
% km =
% 1
% 2
% 3
% 4
% 5
% 6
% 7
% 8
% 9
% 10
km2miles = 0.6214 % km per mile
miles = km2miles*km
% miles =
% 0.6214
% 1.2428
% 1.8642
% 2.4856
% 3.1070
% 3.7284
% 4.3498
% 4.9712
% 5.5926
% 6.2140
km_2_miles_table = [km miles]
% km_2_miles_table =
% 1.0000 0.6214
% 2.0000 1.2428
% 3.0000 1.8642
% 4.0000 2.4856
% 5.0000 3.1070
% 6.0000 3.7284
% 7.0000 4.3498
% 8.0000 4.9712
% 9.0000 5.5926
% 10.0000 6.2140
```

Solution

Add example text here.

.

? Exercise 12.6.1 Oil well resistivity

When drilling for oil, engineers measure the resistivity of the material that they are drilling thru.

A section with a resistivity resistivity > 2 Ohms/m indicates oil or gas.

1. (2 pts) Enter the following array into a MATLAB or Octave script file.

The 1st column is the depth of drilling.

The 2nd column is the resistivity.

```
D_R = [
1.000 0.300
1.010 0.296
1.020 0.298
1.030 0.286
1.040 0.251
1.050 0.280
1.060 0.500
1.070 0.700
1.080 1.000
1.090 2.000
1.100 3.000
1.110 4.692
1.120 4.872
1.130 4.743
1.140 5.070
1.150 4.627
1.160 4.881
1.170 5.396
1.180 4.712
1.190 4.000
1.200 1.900
1.210 1.500
1.220 1.000
1.230 0.962
1.240 0.954
1.250 0.953
1.260 1.002];
```

2. (2 pts) Create a variable named depth = the 1st column's data using array indexing.

3. (2 pts) Create a variable named resistivity = the 2nd column's data using array indexing.

4. (2 pts) Use the find() function to find the indices of the values of resistivity > 2 Ohms/m.

Call this vector of indices oil_index.

5. (2 pts) Use the these indices with the depth vector to create and display a vector of the corresponding oil depths. Call this vector oil_depths

References:

https://www.slb.com/resource-library/oilfield-review/defining-series/defining-log-interpretation

https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/Oil_Rig_NT8.svg/394px-Oil_Rig_NT8.svg.png

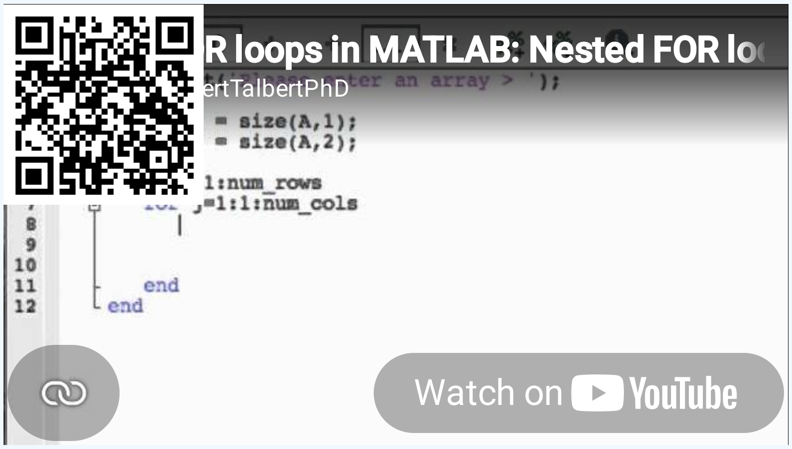
Answer

This is for the student to solve.

.

Exercise 12.6.2 Even-Odd Nested Loops

Watch this video:



```
t('Please enter an array > ');
= size(A,1);
= size(A,2);
1:num_rows
j=1:1:num_cols
    |


    end
end
```

Then code the nested for-loops shown in the video for the even-odd algorithm, with these changes:

1. Delete this line A = input('Please enter an array > '). Instead, code your own 3x4 matrix.

2. Since i is the imaginary number = sqrt(-1), use ii and jj for the loop indices.

3. Test your code.

Answer

Add texts here. Do not delete this text first.

This page titled 12.6: Matrix Tables and Nested Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

12.7: Dot Product

By Carey Smith

The dot product is the sum of the products of the corresponding elements of 2 vectors. Both vectors have to be the same length. Geometrically, it is the product of the magnitudes of the two vectors and the cosine of the angle between them.

```
Figure \(\PageIndex{1}\): a*cos(θ) is the projection of the vector a onto the vecto
```

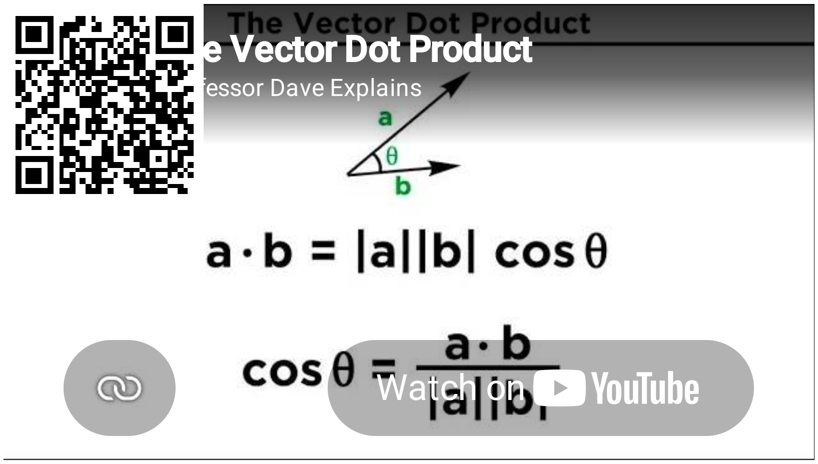
Relationships:

```
a·b = |a|*|b|*cos(θ)
```

```
cos(θ) = |a|*|b| / (a·b)
```

The dot product of a vector with itself is the sum of the squares of the vector's elements. The vector's magnitude (length) is the square root of the dot product of the vector with itself.

This video gives details about dot product:


The Vector Dot Product
e Vector Dot Product
fessor Dave Explains
a
θ
b
$\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta$
$\cos\theta = \frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$
Watch on YouTube

.

Here are examples illustrating the cases of parallel vectors, perpendicular vectors (a.k.a orthogonal), and vectors at 60 degrees relative to each other.

✓ Example 12.7.1 Use dot() to compute the magnitude of a vector

```
a = [1 2 3];
a_dot_a = dot(a,a) % This is the sum of the squares of the elements of A
% 14
a_mag = sqrt(a_dot_a)
% 3.7417
```

MATLAB also has this function to compute the magnitude of a vector:

```
a_mag = norm(a)
```

The norm function computes the square root of the sum of the squares of the elements of the vector.

Solution

Add example text here.

The dot product of 2 different vectors is equivalent to the product of each vector's magnitude (length) times the cos(angle between the 2 vectors).
When the vectors are parallel, the cos = 1, so the dot product = the product of their magnitudes.
When the vectors are perpendicular, the cos = 0, so the dot product = 0.

Example 12.7.2 dot(a,b), a & b parallel

```
a = [1 2 3];
b = 2*a % = [2 4 6]
ab_dot = dot(a,b)
% = 1*2 + 2*4 + 3*6 = 28
|a| = sqrt(1^2 + 2^2 +3^2) = sqrt(14)
|b| = 2*|a| = 2*sqrt(14)
|a|*|b| = 2*14 = 28
```

Solution

Add example text here.

.

Example 12.7.3 dot(c1,d1), c1 & d1 perpendicular: 0 & 90 degs

```
c1 = [1 0 0] % 0 degs
d1 = [0 1 0] % 45 degs
cd1_dot = dot(c1,d1)
% = 0, so these are perpendicular
```

Solution

Add example text here.

.

Example 12.7.4 dot(c2,d2), c2 & d2 perpendicular: +/- 45 degs

```
c2 = [ 1 1 0] % at 45 degs
d2 = [ 1 -1 0] % at -45 degs
cd2_dot = dot(c2,d2)
% = 0, so these are perpendicular
```

Solution

Add example text here.

.

✓ Example $12.7.5$ dot(e1,f1), unit vectors at 0 & 60 degrees

```
e1 = [ 1 0 0] % unit vector at 0 degs
f1 = [ cosd(60) sind(60) 0] % unit vector at 60 degs
% f1 = [0.500 0.866 0]
ef1_dot = dot(e1,f1)
% = 0.500 = cosd(60)
```

Solution

Add example text here.

.

✓ Example $12.7.6$ Not unit vectors at 0 & 60 degrees

```
e2 = 2*[ 1 0 0] % at 0 degs
e2_mag = sqrt(dot(e2,e2)) % = 2
f2 = 5*[ cosd(60) sind(60) 0] % at 60 degs
f2_mag = sqrt(dot(f2,f2)) % = 5

% When they are not unit vectors, we need to divide by their magnitudes.
cos_ef = dot(e2,f2) / (e2_mag*f2_mag)
% = 0.5 = cos(60)
```

Solution

Add example text here.

.

✓ Example $12.7.7$ Use dot product to find the angle between 2 vectors

```
%% dot_product_cos
% Define these 2 vectors:
a = [2, 1.5, 0]
b = [4, 0 ,0]
% Plot the vectors
figure;
plot([0, a(1)], [0, a(2)], 'b', 'LineWidth', 3)
hold on;
plot([0, b(1)], [0, b(2)], 'r', 'LineWidth', 3)
grid on;
axis equal;
legend('a', 'b')

%% Find the angle between them
a_mag = norm(a)
b_mag = norm(b)
cos_ab = dot(a,b)/(a_mag*b_mag)
theta_degs = acosd(cos_ab) % degrees
```

```
%% Draw an arc for the included angle
r = 1; % radius of the arc
ang_vector = linspace(0, theta_degs, 9);
ax = r*cosd(ang_vector);
ay = r*sind(ang_vector);
plot(ax, ay, 'k')
text(1.1, 0.3, '\theta', 'FontSize',16)
```

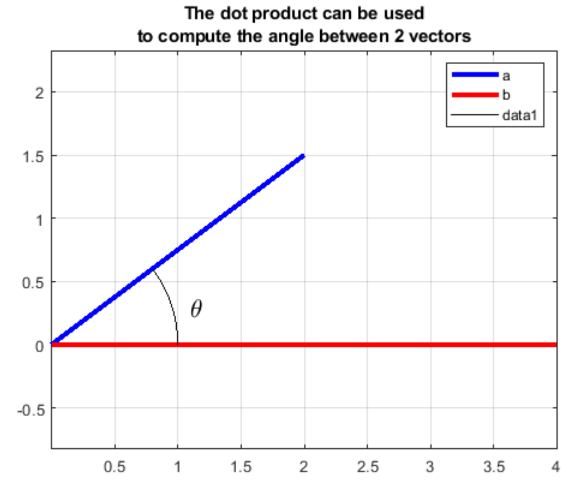


Figure 12.7.2: Dot product can be used to compute the angle between 2 vectors.

Solution

Add example text here.

.

Dot Product in 3 dimensions

The dot product can also be computed for 3-D vectors. Consider:

v1 = [a, b, c]

v2 = [x, y, z]

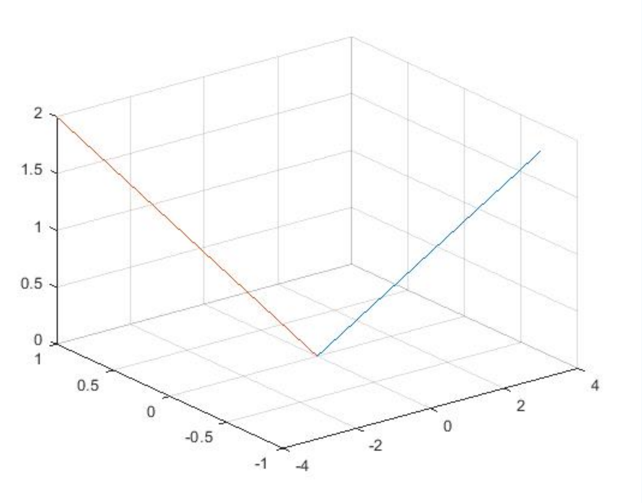
Then dot(v1, v2) = a*x + b*y + c*z

The plot3() function syntax is plot(x, y, z).

Example 12.7.8

```
v1 = [3 -1 2];

v2 = [-4 1 2];

figure;
plot3([0,v1(1)], [0, v1(2)], [0, v1(3)])
grid on;
hold on;
plot3([0,v2(1)], [0, v2(2)], [0, v2(3)])

dp12 = dot(v1,v2)   % = 3*(-4) + (-1)*1 + 2*2 = -9

mag_v1 = sqrt(dot(v1,v1))  % 3.74
mag_v2 = sqrt(dot(v2,v2))  % 4.58
```

```
cos_theta = dp12 / (mag_v1 * mag_v2)  % -0.5249
theta = acosd(cos_theta)   % = 121.7 degrees
```



Solution

Add example text here.

.

✓ Example 12.7.9 Headwind Calculation Using the Dot Product

```
% A jet has heading of alpha = 20 degs.
alpha = 20; % degs
% What is the heading unit vector?
B = [cosd(alpha), sind(alpha), 0] % unit vector
% B = [ 0.9397    0.3420    0]
% The wind is 30 km/s at an angle of 150 degs. So its vector is
wind_vel = 30; % km/s
wind_ang = 150 % degs
A = wind_vel*[cosd(wind_ang), sind(wind_ang), 0]
% A = [-25.9808  15.0000   0]
% What is the component of wind along the jet’s heading?
%Answer:
head_wind = dot(A, B)
% head_wind = -19.2836
```

Solution

Add example text here.

.

This page titled 12.7: Dot Product is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

12.8: Cross Product and Torque

By Carey Smith

Cross product calculations are inherently 3-dimensional. The cross product of 2 vectors, a and b, is another vector, c, which is perpendicular to both a and b.

When a and b are parallel, c is zero.

When a and b are perpendicular, the magnitude of c = the product of the magnitudes of a and b.

Watch this video!

It gives a visual explanation of the cross product, including the right-hand rule. It also demonstrates the dot product.

oss Product and Dot Product: Visu
sics Videos by Eugene Khutoryansky
Video by: Eugene Khutoryansky
Narrator and dialogue editor: Kira Vincent
Watch on YouTube

.

Examples for parallel vectors, perpendicular vectors, and vectors at a 60 degree angle are given here. 3-D Plots are given for each example below, except for the zero case. These 3-D plots use the plot3() function. The plot3() function syntax is plot(x, y, z).

✓ Example 12.8.1 cross(a,b) **a & b parallel**

The cross product of parallel vectors is zero.

a = [1 2 3]
b = 2*a % = [2 4 6]
ab_cross = cross(a,b) % = [0 0 0]

Solution

Add example text here.

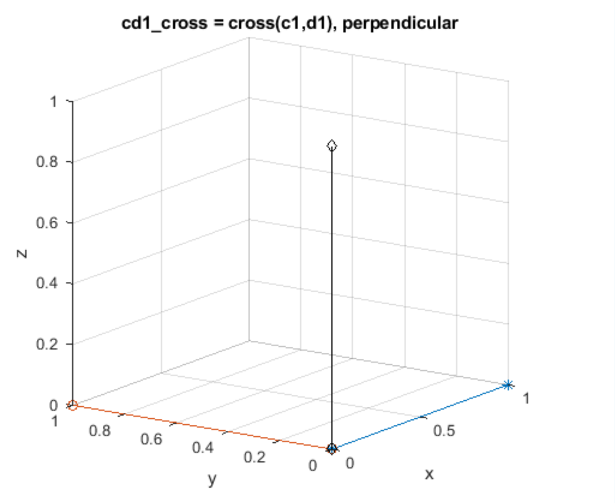
.

✓ Example 12.8.2 cross(c1,d1) c1 & d1 perpendicular unit vectors

c1 = [1 0 0]
d1 = [0 1 0]
cd1_cross = cross(c1,d1)
% = [0 0 1]

```
% Plot each vector
figure;
plot3([0,c1(1)],[0,c1(2)],[0,c1(3)],'-*')
hold on;
plot3([0,d1(1)],[0,d1(2)],[0,d1(3)],'-o')
% Plot the cross-product vector on the same figure
plot3([0,cd1_cross(1)],[0,cd1_cross(2)],[0,cd1_cross(3)],'k-d')
axis equal
grid on;
xlabel('x')
ylabel('y')
zlabel('z')
% Use tools->Rotate 3D to verify that cd1_cross is orthogonal
% to the 1st 2 vectors and follows the right-hand rule.
title('cd1\_cross = cross(c1,d1), perpendicular')
```

Solution



Add example text here.

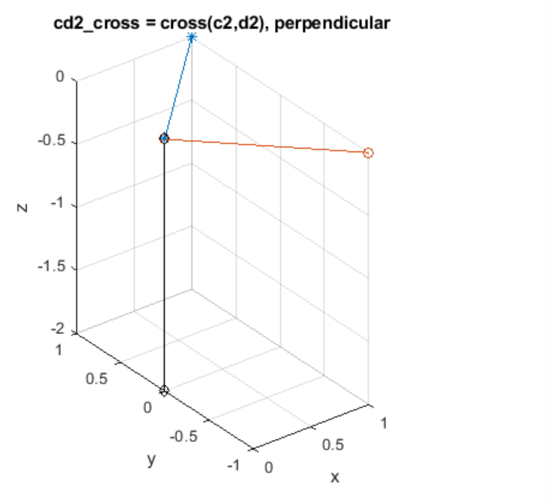
.

Example 12.8.3 cross(c2,d2) c2 & d2 perpendicular, 45 degs

```
c2 = [ 1 1 0] % at 45 degs
d2 = [ 1 -1 0] % at -45 degs
cd2_cross = cross(c2,d2) % = [0 0 -2]
% Plot each vector
figure;
plot3([0,c2(1)],[0,c2(2)],[0,c2(3)],'-*')
hold on;
plot3([0,d2(1)],[0,d2(2)],[0,d2(3)],'-o')
% Plot the cross-product vector on the same figure
plot3([0,cd2_cross(1)],[0,cd2_cross(2)],[0,cd2_cross(3)],'k-d')
axis equal
grid on;
xlabel('x')
```

```
ylabel('y')
zlabel('z')
% Use tools->Rotate 3D to verify that cd2_cross is orthogonal
% to the 1st 2 vectors and follows the right-hand rule.
title('cd2\_cross = cross(c2,d2), perpendicular')
```

Solution



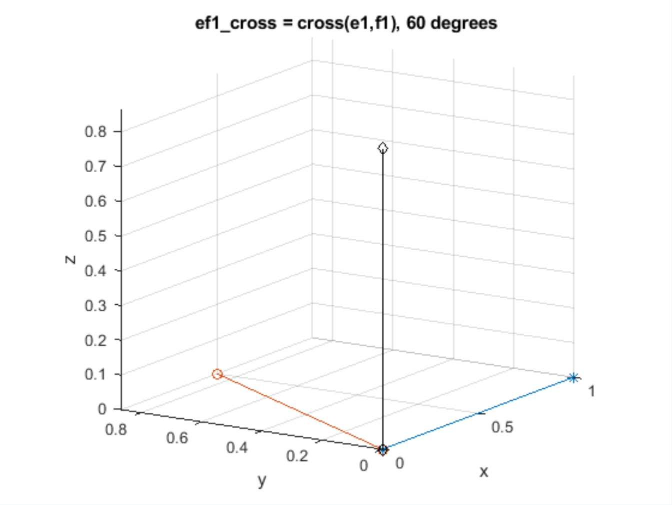
Add example text here.

.

Example 12.8.4 cross(e,f) angle between e and f = 60 degrees

```
e1 = [1 0 0] % at 0 degs
f1 = [cosd(60) sind(60) 0] % at 60 degs
% f = [0.500 0.866 0]
ef1_cross = cross(e1,f1)
% = [0 0 0.866]
% Plot each vector
figure;
plot3([0,e1(1)],[0,e1(2)],[0,e1(3)],'-*')
hold on;
plot3([0,f1(1)],[0,f1(2)],[0,f1(3)],'-o')
% Plot the cross-product vector on the same figure
plot3([0,ef1_cross(1)],[0,ef1_cross(2)],[0,ef1_cross(3)],'k-d')
axis equal
grid on;
xlabel('x')
ylabel('y')
zlabel('z')
% Use tools->Rotate 3D to verify that ef1_cross is orthogonal
% to the 1st 2 vectors and follows the right-hand rule.
title('ef2\_cross = cross(e1,f1), 60 degrees')
```

Solution



Add example text here.

.

Torque

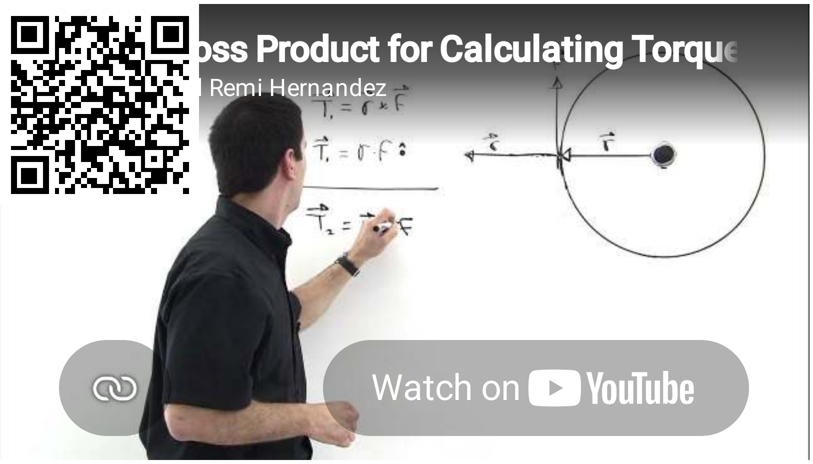
For a lever of length R, and a Force, F, Torque = RxF (R cross F).

When the Force is in pounds and the length in feet, then the units are pound-feet.

When the Force is in Newtons and the length in meters, then the units are Newton-meters.

In mechanical engineering, torque is known as "moment".

This video illustrates torque nicely: (Cross Product for Calculating Torque, http://BigBangPhysics.com)



Exercise 12.8.1 Torque HW

Consider a wrench loosening a bolt..
Our coordinate system is:

- x is horizontal in the plane of the wrench
- y is horizontal perpendicular to x (into the paper)
- z is vertical

[2 pts] Set up the force and wrench vectors:

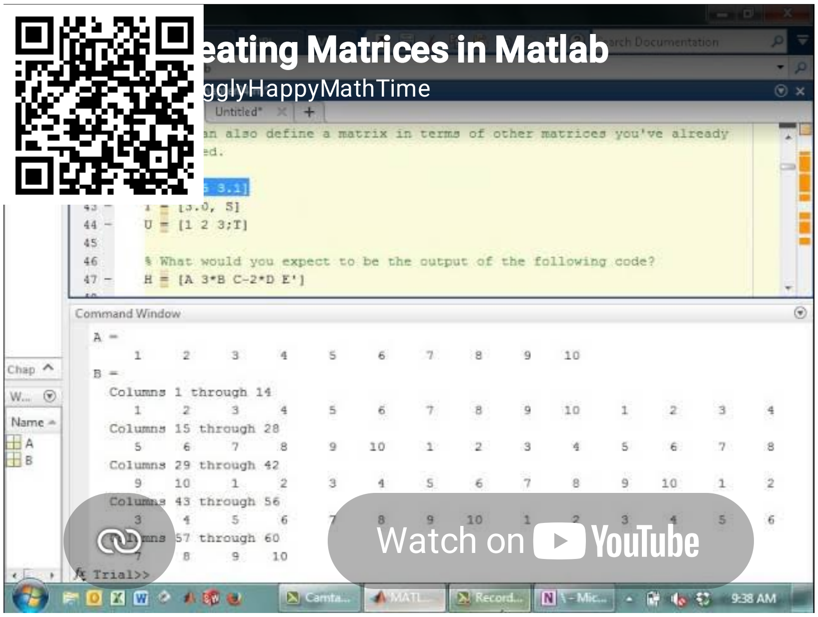
> A force is applied near the end of the wrench at a distance R = 0.20m from the bolt on the left side at a 20 degree angle from horizontal. When the bolt turns, it will have a counter-clockwise rotation.
> R = 0.20*[-cosd(20), 0, sind(20)] %(m)
> The magnitude of the force is 100N. The direction of the force is down.
> F = 100*[0, 0, -1] %(N)
>
> [4 pts] Compute the torque using the cross() function.
> torque = cross(R,F)
>
> [4 pts] Plot the R, F, and torque vectors on 1 figure using code similar to the cross product examples.
> Since the units of R, F, and torque are all different, do not use "axis equal".
> Add title('Torque')
> Use tools->Rotate 3D to verify that the cross product is orthogonal to the R and F vectors and follows the right-hand rule
>
> **Answer**
>
> Add texts here. Do not delete this text first.

.

This page titled 12.8: Cross Product and Torque is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

12.9: Chapter 12 Matrix Videos English and Espanol

English


eating Matrices in Matlab
gglyHappyMathTime
Watch on YouTube

.

R loops in MATLAB: Nested FOR l
ertTalbertPhD
Watch on YouTube

Espanol

Matrices y operaciones I - 6 - Octave

atrices y operaciones I - 6 - Octave
siored
Matrices
Operaciones
Watch on YouTube

Matrices y operaciones II - 7 - Octave

atrices y operaciones II - 7 - Octave
siored
Matrices
Operaciones II
Watch on YouTube

Operador dos puntos - 9 - Octave

erador dos puntos - 9 - Octave
siored
perador :
Watch on YouTube

ATLAB PARA ALGEBRA LINEAL 01
as
IA
TLAB
CON ENFASIS
EN ALGEBRA LINEAL
Watch on YouTube

This page titled 12.9: Chapter 12 Matrix Videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

13: Linear Algebra

13.1: Basic Matrix Multiplication
13.2: Matrix Multiplication Applications
13.3: Matrix Inverse, Rank and Determinant
13.4: Solving Systems of Equations
13.5: Systems of Equations Examples and Exercises
13.6: Over-Determined Systems of Equations
13.7: Chapter 13 Linear Algebra Videos in English and Espanol

---

This page titled 13: Linear Algebra is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

13.1: Basic Matrix Multiplication

By Carey Smith

13.1.1: Matrix Multiplication Summary:

A matrix times a vector (3x3 case):

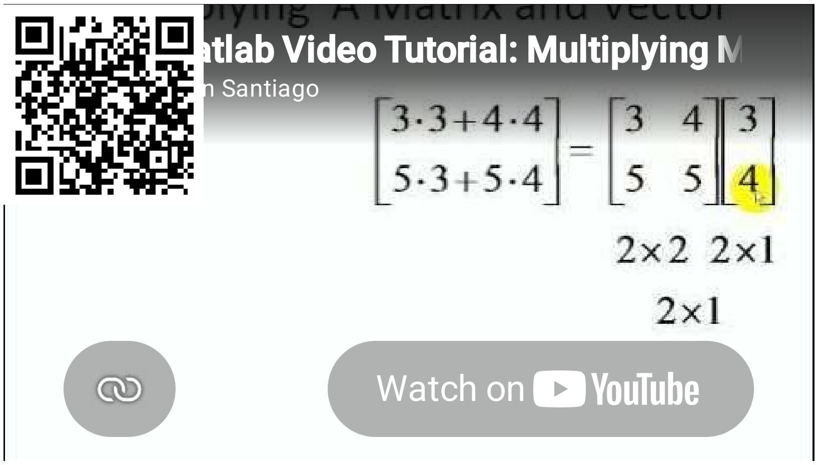
$$\begin{array}{l} \quad\;\;\, |a \quad b \quad c| \quad |x| \quad |a*x \quad b*y \quad c*z| \\ A*B = |p \quad q \quad r| * |y| = |p*x \quad q*y \quad r*z| \\ \quad\;\;\, |u \quad v \quad w| \quad |z| \quad |u*x \quad v*y \quad w*z| \end{array}$$

Matrices do not need to be square, but the number of columns in A needs to equal the number of rows in B.

Matrix multiplication is generally not commutative. That is A*B ≠ B*A, in general.

.

Watch this video about multiplying matrices:


atlab Video Tutorial: Multiplying M
n Santiago
$\begin{bmatrix}3\cdot3+4\cdot4\\5\cdot3+5\cdot4\end{bmatrix}=\begin{bmatrix}3&4\\5&5\end{bmatrix}\begin{bmatrix}3\\4\end{bmatrix}$
2×2 2×1
2×1
Watch on YouTube

Matlab Video Tutorial: Multiplying Matrices and Vectors, https://www.youtube.com/watch?v=sgzFn42jU6Y

.

Example 13.1.1 Matrix Multiply 3x3 times a vector

```
A = [3 2 1;

2 1 0

1 0 -1]

B = [4

5

6]

A*B
```

Solution

```
A*B =

28
13
-2

% = [28 %(28 = 3*4 + 2*5 + 1*6)
% 13 %(13 = 2*4 + 1*5 + 0*6)

% -2] %(-2 = 1*4 + 0*5 + -1*6)
```

.

Exercise 13.1.1 Matrix Multiplication True/False

1. True or False: These 2 matrices (which have 3 rows and 2 columns) can be multiplied together:

```
     [ 1 5 ]      [ 9 5 ]
A = | 2 9 | B = | 8 9 |
     [ 3 4 ]      [ 7 5 ]
```

2. True or False: These 2 matrices can be multiplied together:

```
     [ 1 5 ]
A = | 2 9 |; B = [ 1 3 2 ]
     [ 3 4 ] [ 4 6 9 ]
```

Answer

1. No, because the number of columns in A (2) does not equal the number of rows in B (3).
1. Yes, because the number of columns in A (2) does equal the number of rows in B (2).

Exercise 13.1.2 Matrix Multiplication Calculation

1. What is A*B ?

```
A = [1 4
      3 -5]

B = [ 2 12

       3 1]
```

(Do a matrix multiply, not an element-by element multiply)

Answer

```
A*B = [14 16

        -9 31]
```

.

A matrix with ones on the diagonal and zeros elsewhere is called the Identity matrix:

```
I = eye(3) = [1 0 0

             0 1 0

             0 0 1]

I*A = A

A*I = A
```

For more information, see: https://en.Wikipedia.org/wiki/Matrix_multiplication

.

This page titled 13.1: Basic Matrix Multiplication is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

13.2: Matrix Multiplication Applications

By Carey Smith

13.2.1: Rotation Matrices

One use of matrices is to rotate vectors.

A 2D Rotation matrix is of this form:

Mrot2D = [cosd(▯) -sind(▯)]

[sind(▯) cosd(▯)]

(Use cosd(▯) and sind(▯) when the angle is in degrees.)

Use cos(▯) and sin(▯) when the angle is in radians.)

Example 13.2.3 2D Rotate by 60 degrees clockwise:

Mrot2D60 = [cosd(60) -sind(60)] = [0.5 -0.866]

[sind(60) cosd(60)] [0.866 0.5]

Example code is in the attached file Ex_Rotation2D.m

Solution

Add example text here.

3D Rotation matrices

A 3D Rotation matrix about the z-axis is has a 2D rotation sub matrix in it. It is of this form::

Mrot3Dz = [cosd(▯z) -sind(▯z) 0]

[sind(▯z) cosd(▯z) 0]

[0 0 1]

3D Rotation matrix about the X-axis is:

Mrot3Dx = [1 0 0]

[0 cosd(▯x) -sind(▯x)]

[0 sind(▯x) cosd(▯x)]

3D Rotation matrix about the Y-axis is:

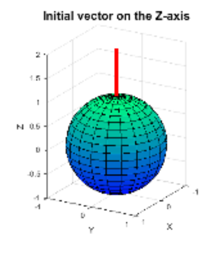
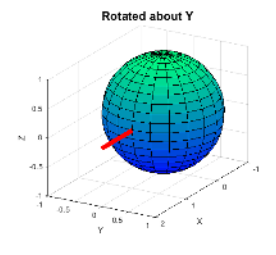
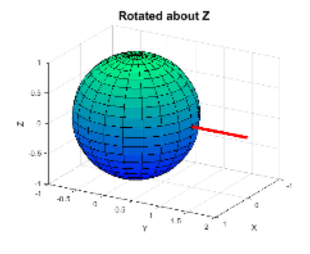
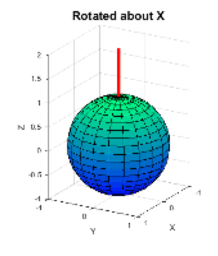
Mrot3Dy = [cosd(▯y) 0 sind(▯y)]

[0 1 0]

[-sind(▯y) 0 cosd(▯y)]

Example code is in the attached file Ex_Rotation3D.m

This produced the following figures:

			
Initial vector is on the Z-axis	Rotate by+90 about Y it goes to the X-axis	Rotate by+90 about Z it goes to the Y-axis	Rotate by+90 about X it goes to the Z-axis

13.2.2: Matrix Powers

A matrix can be raised to a power using matrix multiplication.

Example 13.2.4

A = [1 2
3 4]

A^2 = A*A = [1 2]*[1 2] = [7 10]

[3 4] [3 4] [15 22]

A^4 = [199 290]
[435 634]

Solution

Add example text here.

13.2.3: Life Cycle Matrix

A Matrix can be used to model the number of plants or animals at various stages from year to year.

Example 13.2.5 Draba Plant Matrix Life Cycle

This example considers a small flowering plant called Lake Tahoe Draba. Information about this plant and an image is available at: https://calscape.org/Draba-asterophora-(). That image is copyrighted. The image below is a creative commons image of a similar draba species. The numbers used in this example are made up simply to illustrate a possible application of matrix multiplication.

https://commons.wikimedia.org/wiki/F...orullensis.jpg, Tim Park

This file is licensed under the Creative Commons Attribution-Share Alike 4.0 International license.

Consider these stages of a plant:

- Seed
- Seedling (1st year)

- Mature plant

The numbers of plants in each stage will be the elements of a vector, D, for Draba.

Initialize D to have 20 mature plants, with zeros for the other stages:

```
D1 = [0
      0
      20]
```

Let T = the transition matrix from 1 year to the next.

Assume the following average numbers: (These are made up)1 plant produces 8 seeds.

Fraction of seeds that become seedlings = 0.2

Fraction of seedlings that become mature plants = 0.5

Fraction of plants that survive from year to year = 0.7

Create the transition matrix's rows:

seeds = 8*number of mature plants:

-> `T_seeds = [0 0 8]`

seedlings = 0.2*number of mature seeds

-> `T_seedlings = [0.2 0 0]`

New mature plants from seedling = 0.5*seedlings

0.7*mature plants survive from the previous year

```
T_mature = [0 0.5 0.7]
```

The T matrix is created by combining these 3 vectors:

```
T = [T_seeds
T_seedlings
T_mature ]
```

```
T= [0   0   8
    0.2 0   0
    0   0.5 0.7 ]
```

Compute the numbers of each stage in year 2:

```
D2 = T*D1
```

Compute the numbers of each stage in year 3:

```
D3 = T*D2
```

Compute the numbers of each stage in year 4:

```
D4 = T*D3
```

Compare D4 another way using a matrix power:

```
D4b = (T^3)*D1
```

Verify that this produces the same result.

Compute the numbers of each stage in year 10:

```
D10 = (T^9)*D1
```

Solution

Add example text here.

.

This page titled 13.2: Matrix Multiplication Applications is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

13.3: Matrix Inverse, Rank and Determinant

By Carey Smith

Consider this square matrix:

```
B = [1 2 3;
     4 5 6;
     7 8 0]
```

The inverse of B, IB, is a matrix such that B*IB = the identity matrix and IB*B = the identity matrix.

Not every square matrix has an inverse. For example these 3x3 matrices do not have inverses:

```
C = [1 0 0;

     0 0 0;

     0 0 0]

D = [1 0 0;

     0 2 3;

     1 2 3]
```

A matrix which does not have an inverse is called a "singular" matrix.

The rank of a matrix is the number of independent rows.
When the rank of a square matrix = the number of rows, it has "full rank" and is non-singular, so it has an inverse.

The rank of a matrix can be computed with the MATLAB function rank():

```
B_rank = rank(B) % = 3. Since B has 3 rows and columns and its rank is 3, B has full rank and is non-singular.

C_rank = rank(C) % = 1. C does not have full rank.

D_rank = rank(D) % = 2. D does not have full rank.
```

Another characteristic of a square matrix is its determinant.
When the determinant is not 0, it has full rank. It doesn't matter whether the determinant is positive or negative. When the determinant is not 0, the matrix is nonsingular and can be inverted. The calculation of the inverse divides by the matrix determinant, which is why it can't be zero.

The determinant of a matrix can be computed with the MATLAB function det():

```
B_det = det(B) = 27 => B is nonsingular and can be inverted.

C_det = det(B) = 0 => C is singular and cannot be inverted.

D_det = det(B) = 0 => C is singular and cannot be inverted.
```

The inverse of a square matrix can be computed with the MATLAB function inv():

```
W = inv(B)

% -1.7778  0.8889 -0.1111
%  1.5556 -0.7778  0.2222
% -0.1111  0.2222 -0.1111

Test the inverse:
IB1 = B*W
% 1 0 0
% 0 1 0
% 0 0 1
```

This is the 3x3 identity matrix

Although matrix multiplication is not commutative in general, multiplying a matrix by its inverse is commutative. The same result occurs when the order of the matrix and its inverse is reversed.
`IB2 = W*B` computes the same result.

✓ Example 13.3.1 Inverse of 3x3 Pacal matrix

```
C = pascal(3)
```

Compute the rank and determinant of C.

If it is non-singular, compute its inverse, C_inv/

Then test that C_inv*C is the identity matrix.

Solution

```
det_C = det(C) % 1 Non-Singular
C_rank = rank(C) % 3 Non-Singular
C_inv = inv(C)
C_inv*C
```

.

✓ Example 13.3.2 Singular matrices

Compute the rank and determinant of these singular matrices:

```
%% Singular Matrix #1
% 2nd row is a multiple of the 1st row,
% so it is not independent
S1 = [ 1 2 3
.      2 4 6
.      7 8 9]

S2 = [ 1 2 3
       4 5 6
       5 7 9]

S3 = [ 1 0 0
       1 0 0
       0 0 0]
```

Solution

Add example text here.

For more information, read https://en.Wikipedia.org/wiki/Invertible_matrix

? Exercise 13.3.1 Operations on a "Hilbert" Matrix

By Troy Siemers

Consider the 5 x 5 \Hilbert matrix"

```
 H5 = [1   1/2 1/3 1/4 1/5;
       1/2 1/3 1/4 1/5 1/6;
       1/3 1/4 1/5 1/6 1/7;
       1/4 1/5 1/6 1/7 1/8;
       1/5 1/6 1/7 1/8 1/9];
```

a. Find the determinant of H5.
b. Find the inverse of H5.
b. Find the transpose of H5.
c. Using the commands in the text, find the dimensions of H5, the column sums of H5, and the row sums of H5.
d. Use the max function to determine the value of the maximum entry of H5.
e. Find the eigenvalues and eigenvectors of H5.
f. Find the matrices H5^2, H5.^ 2, and H5./H5 and explain your answers.
g. By referencing H5, create a matrix that consists of the 2nd and 3rd rows of H5 (Do not simply retype the entries).

Answer

Add texts here. Do not delete this text first.

.

This page titled 13.3: Matrix Inverse, Rank and Determinant is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

13.4: Solving Systems of Equations

This is an edited version of Siemers section 2.5.

We can use matrices to solve systems of linear equations. It is a good idea to read up a bit on some matrix algebra.
Suppose we have the following system of equations:

$$\left\{\begin{array}{c}3x+2y-z=10\\ -x+3y+2z=5\\ x-y-z=-1\end{array}\right\}$$

We will solve this system, i.e. find the values of the variables that satisfy all of the equations simultaneously, in two ways: using matrix inverses and using “left division.”

Method 1: **Using the matrix inverse**

Here we create two matrices, one for the coefficients of the variables and one for the constants to the right of the equals signs.

```
Coeffs = [ 3  2 -1;
          -1  3  2;
           1 -1 -1];
Constants = [10;
              5;
             -1];
```

Then the matrix form of the system of equations is:

```
Coeffs * xyz = Constants
```

Where the solution is the vector xyz:

```
xyz = [x

       y

       z]
```

We see that it would be nice to divide both sides by `Coeffs`. The equivalent operation for matrices is to multiply both sides by the matrix inverse of the Coeffs matrix. Before using the inverse of the Coeffs matrix, we need to verify that the matrix is invertible by computing its determinant:

```
Coeffs_det = det(Coeffs)  % =1
```

Since the determinant of Coeffs is non-zero, we can solve the system with the inverse:

```
xyz1 = inv(Coeffs)*Constants
```

```
% xyz1 =
%   -2.0000
%    5.0000
%   -6.0000
```

This tells us that the only solution is x = -2, y = 5, z = -6.

Method 2: **Using left division**

The motivation for this method is complicated. The algorithm is Gaussian elimination, which is not actually a division, but that a division symbol is used by MATLAB to apply this algorithm, as shown below. (If you are interested in the details, you can read the Matlab documentation on left (and right) division of matrices.) We use the same two matrices as in Method 1 and use the backslash \ symbol (be careful to use the correct slash):

```
>> xyz2 = Coeffs\Constants
```

```
% xyz2 =
%   -2.0000
%    5.0000
%   -6.000
```

This also tells us that the only solution is `x = -2, y = 5, z = -6.`

Usually methods 1 & 2 produce nearly identical solutions. Mathworks says: "x = A\b is computed differently than x = inv(A)*b and is recommended for solving systems of linear equations." (https://www.mathworks.com/help/matlab/ref/inv.html)

This page titled 13.4: Solving Systems of Equations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Troy Siemers (APEX Calculus) via source content that was edited to the style and standards of the LibreTexts platform.

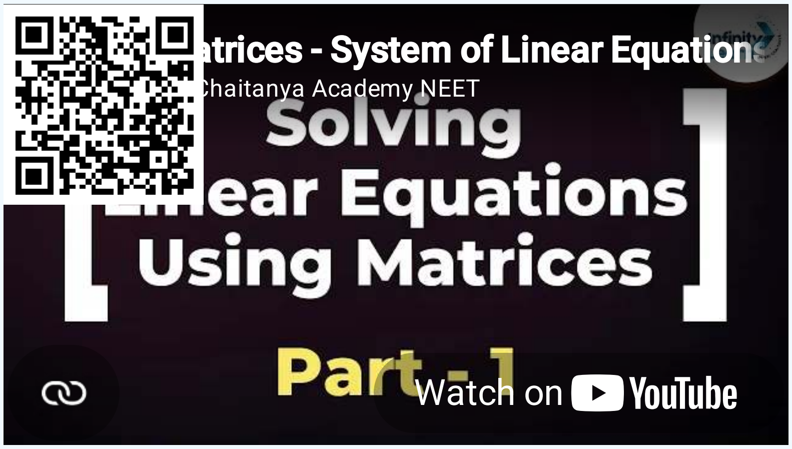
- **2.5: Systems of Equations** by Troy Siemers is licensed CC BY-NC 3.0. Original source: https://drive.google.com/file/d/0B2Pdg3LqVPNPLUhLbks0QzF5ZVhmWGp3YW5hYlVBd3RDN2x3/view?resourcekey=0-fbyowylmkO1jBbS2ehDP0g&usp=sharing.

13.5: Systems of Equations Examples and Exercises

By Carey A. Smith

Exercise 13.5.1 : Solving Linear Equations Video and Homework

Watch this 4 minute video. Then code up the example shown and solve using Matlab's inv() function.


atrices - System of Linear Equations
Chaitanya Academy NEET
Solving
Linear Equations
Using Matrices
Part - 1
Watch on YouTube

Helps:

(1) Define the matrix A shown in the video.

(2) Define the vertical vector B shown in the video.

(3) X is the unknown vertical vector whose elements are x and y

Answer

X = inv(A)*B

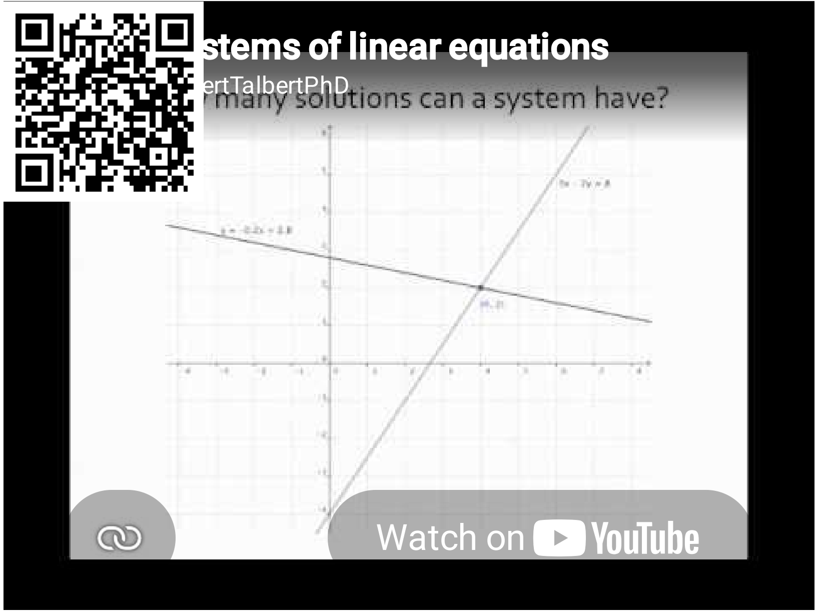
Test your answer by computing:

B2 = A*X

You should get B2 == B

.

13.5.1: This video shows how to solve a system of equations graphically.


stems of linear equations
ertTalbertPhD
many solutions can a system have?
Watch on YouTube

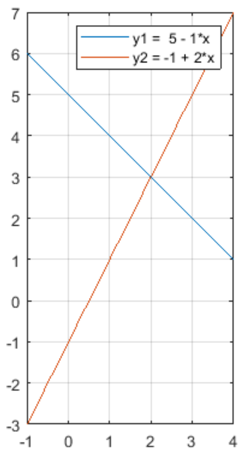
.

Example 13.5.1 : 2 equations, 2 unknowns Graphic and Matrix methods

```
1*x + y = 5
-2*x + y = -1
```

1a. Graphical Solution

Each of 2 equations in 2 unknowns, x and y, can be plotted as a line. Their intersection is a graphical solution of the 2 equations. The equations and their plot is shown here. The intersection is the point (2, 3). So the solution of this system of equations is x=2, y = 3.



1b. Solution using the matrix inverse.

Matrix formulation:

```
% A*xy = B
A = [ 1 1
-2 1]
```

```
% xy = [x
% y]
B = [ 5
-1]
rank(A) % 2 (full)
det(A) % 3 (non-zero)
% Method 1: Solution using the inverse of A:
% inv(A)*A*xy = inv(A)*B
% inv(A)*A = I, so
xy = inv(A)*B
```

```
% Solution = (2, 3), same as above
```

1b. Solution using left matrix-division .

Same formulation as in part 1b. The left matrix-division conceptual derivation is:

```
A*xy = B
(1/A)(A*xy) = (1/A)*B [Not B/A--not commutative]
```

```
Write (1/A)*B as xy = A\B
```

```
% Solution = (2, 3), same as above
```

Solution

The solutions were given in the example.

Important note about left matrix-division:

xy = A\B is “Left Matrix-Division”. It solves the system of equations by Gaussian Elimination, which is a series of matrix operations. When A is a 3x3 matrix and B is a 3x1 vector, then the result, xy, is a 3x1 vector.

Unfortunately, MATLAB has another function, ldivide(), which sounds like it is the same thing, but it is not. Do not use ldivide(); it does an element-by-element divide, rather than a matrix operation. When A is a 3x3 matrix and B is a 3x1 vector, then the result of using ldivide(), xy is a 3x3 matrix, which is not the solution of the system of equations.

So use either

xy = A\B

or

xy = inv(A)*B

.

? Exercise 13.5.2 : System_of_Equations_HW2

Given this system of equations:

1*x + 1*y + 2*z = 11.1
0*x -1*y + 3*z = 5.4
4*x + 2*y + 1*z = 9.7

Solve for x, y, and z using the following steps:

Part 1 (3 pts). Create the matrix, M, of the coefficients.

Part 2. (1 pt). Create the right-hand side vector of constants. Call it R.

Part 3. (1 pt). Compute the determinant and verify that it is not 0

Part 4. (3 pts) Let xyz = the vector of unknown values of x, y, and z. Solve for xyz using the inverse of M.

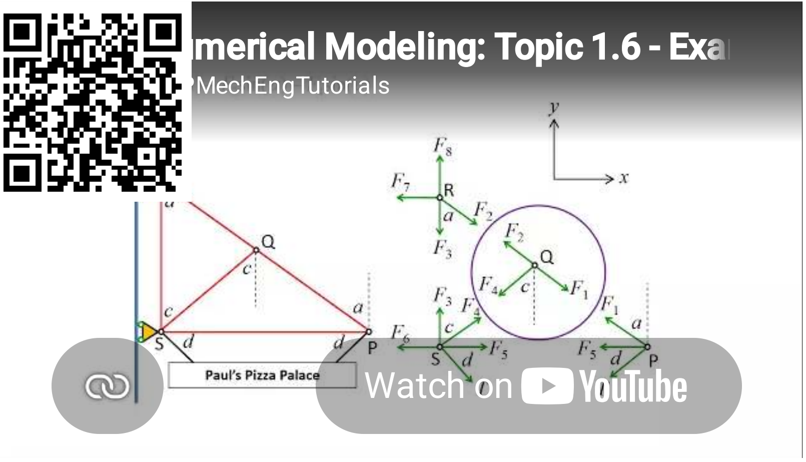
Part 5: (3 pts) Solve for xyz using the inverse of M, using left matrix-division.

Part 6: (1 pt) Multiply M*xyz and verify that it equals R.

Answer

The answer is not given here.

13.5.2: .
This video shows how systems of equations are used to solve a structural engineering truss problem:


merical Modeling: Topic 1.6 - Exa
MechEngTutorials
Paul's Pizza Palace
Q
R
S
P
a
c
d
F_1
F_2
F_3
F_4
F_5
F_6
F_7
F_8
y
x
Watch on YouTube

.

Example 13.5.2 : Electric_Circuit_KVL1

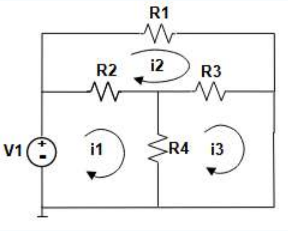
You do not need to to be a circuits expert. The equations needed will be given to you. Then these will be put in the form of a system of linear equations that can be solved.

Kirkoff's Voltage Law:

The sum of the voltages around any loop without a voltage source = 0.

The sum of the voltages around any loop with a voltage source = the voltage of the source.

When a resistor is part of 2 loops, the sum of the currents across the resistor needs to be used. We assume that the current is positive in the direction of the arrows in the drawing. When the direction of the loop arrow is backwards through a resistor, then use the negative of voltage across the resistor.


R1
R2
i2
R3
V1
i1
R4
i3

The voltage supply is V1 = 3 volts.

The resistors are:

R1 = 1 Ohm

R2 = 2 Ohms

R3 = 3 Ohms

R4 = 4 Ohms

The expressions for the voltages across each resistor are:

We use the formula: V = R*I, where V = voltage, R = resistance, I = current.

VR1 = R1*I2 %Voltage across R1 (left to right)
VR2 = R2*(I1 - I2) %Voltage across R2 (left to right)
VR3 = R3*(I3 - I2) %Voltage across R3 (left to right)
VR4 = R4*(I1 - I3) %Voltage across R4 (top to bottom)

The voltages around each loop are:

```
R2*(I1 - i2) + R4*(I1 – i3) = V1 %Loop1
```

```
R1*I2 - R3*(I3 - I2) - R2*(I1 - I2) = 0 %Loop2
```

```
R3*(I3 - I2) - R4*(I1 - I3) = 0 %Loop3
```

Since the unknowns are the currents, I1, I2, I3, we combine like terms, so that we can put this in matrix form:

```
(R2+R4)*I1 (-R2)*i2 + (-R4)*i3 = V1 %Loop1
(-R2)*I1 + (R1+R2+R3)*i2 + (-R3)*i3 = 0 %Loop2
(-R4)*I1 + (-R3)*i2 + (R3+R4)*i3 = 0 %Loop3
```

The matrix form is coef*I123 = vconst, where:

```
coef = [(R2+R4), (-R2), (-R4);
(-R2), (R1+R2+R3), (-R3);
(-R4), (-R3), (R3+R4)]
```

I123 = the vector of the unknown currents.

```
vconst = [V1;
0;
0]
```

The computed value of matrix coef is:

```
% 6 -2 -4
% -2 6 -3
% -4 -3 7
```

det(coef) % = 26, so coef is non-singular.

We can now solve for I123 using either the inverse of coef or using left-matrix division.

Once we have computed I123, we can compute the currents in each loop and the voltages across each resistor.

Solution

Using the inverse of coef:

I123 = inv(coef)*vconst
% 3.8077
% 3
% 3.4615

or

Using left matrix division:
I123 = coef\vconst %Same answer to 5 significant figures

Check the answer by computing the matrix form:
check1 = coef*I123

This equals the right-hand-side, vconst.

We can use I123 to compute the currents in each loop:

I1 = I123(1)
I2 = I123(2)
I3 = I123(3)

We can use these currents to compute the voltages across each resistor:

VR1 = R1*i2 %Voltage across R1 (left to right)
VR2 = R2*(I1 - i2) %Voltage across R2 (left to right)
VR3 = R3*(i3 - i2) %Voltage across R3 (left to right)
VR4 = R4*(I1 - i3) %Voltage across R4 (top to bottom)

Then we can verify that the sums of the voltages around each loop are essentially zero:
VLoop1 = -V1 + VR2 + VR4
VLoop2 = VR1 - VR3 - VR2
VLoop3 = VR3 - VR4

.

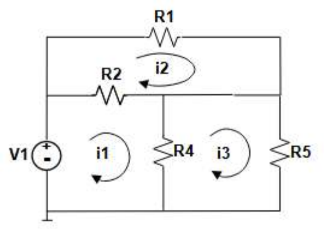
Exercise 13.5.3 : Electric_Circuit_KVL2

Kirkoff’s Voltage Law:

The sum of the voltages around any loop without a voltage source = 0.

The sum of the voltages around any loop with a voltage source = the voltage of the source.

This circuit is similar to the previous example. R3 has been removed and R5 has been added. The values of the voltage and the resistances have been changed.



The voltage supply is V1 = 6 volts.

The resistors are:

R1 = 12 Ohms

R2 = 6 Ohms

R4 = 3 Ohms

R5 = 6 Ohms

Use this formula: V = R*I, where V = voltage, R = resistance, I = current.

The expressions for the voltages across each resistor are:

VR1 = R1*I2 %Voltage across R1 (left to right)
VR2 = R2*(I1 - I2) %Voltage across R2 (left to right)
VR4 = R4*(I1 - I3) %Voltage across R4 (top to bottom)
VR5 = R5*I3 %Voltage across R5 (top to bottom)

The voltages around each loop are:

```
R2*(I1 - I2) + R4*(I1 - I3) = V1 %Loop1
R1*I2 - R2*(I1 - I2) = 0 %Loop2
```

```
R5*I3 - R4*(I1 - I3) = 0 %Loop3
```

Since the unknowns are the currents, I1, I2, I3, we combine like terms, so that we can put this in matrix form:

```
(R2+R4)*I1 (-R2)*i2 + (-R4)*i3 = V1 %Loop1
(-R2)*I1 + (R1+R2+R3)*i2 + (-R3)*i3 = 0 %Loop2
(-R4)*I1 + (-R3)*i2 + (R3+R4)*i3 = 0 %Loop3
```

The matrix form is coef2*I123 = vconst, where:

```
coef2 = [(R2+R4), (-R2), (-R4);
(-R2), (R1+R2+R3), (-R3);
(-R4), (-R3), (R3+R4)]
```

I123 = the vector of the unknown currents.

```
vconst2 = [V1;
0;
0]
```

Compute the determinant of the coef2 matrix.

Compute I123 using either the inverse of coef2 or using left-matrix division.

Check the computed value of I123 by computing the matrix form:
check2 = coef2*I123

This should equal the right-hand-side, vconst2.

Use I123 to compute the voltages across each resistor:

VR1 = R1*i2 %Voltage across R1 (left to right)
VR2 = R2*(I1 - i2) %Voltage across R2 (left to right)
VR3 = R3*(i3 - i2) %Voltage across R3 (left to right)
VR4 = R4*(I1 - i3) %Voltage across R4 (top to bottom)

Verify that the sums of the voltages around each loop are essentially zero:
VLoop1 = -V1 + VR2 + VR4
VLoop2 = VR1 - VR3 - VR2
VLoop3 = VR3 - VR4

Answer

The answer is not given here.

.

Exercise 13.5.4 Metal Alloy System of Equations

An engineer has samples of 3 Magnesium-Aluminum-Zinc alloys.

These alloys are named Az63, Az92, and Az92 (The 2 numbers represent the percentages of Aluminum and of Zinc)

% The fractions of each metal in each alloy are given in this table:

```
% | AZ63 | AZ91 | AZ92 |
% ---|------|------|------|
% Mg | 0.91 | 0.90 | 0.89 |
% Al | 0.06 | 0.09 | 0.09 |
% Zn | 0.03 | 0.01 | 0.02 |
```

Part 1 (3 pts). Create M3 = 3x3 coefficient matrix with these proportions.

Part 2: (1 pt). Compute the determinant of M3. Since the numbers in the table are small, we expect the determinant to be on the order of 10e-4. For this problem, as long as it is > 10e-8, then the matrix is not singular and can be properly inverted.

Part 3 (2 pts). The engineer wants to combine one sample of each of the 3 alloys, such that the combination has the following weights of each metal (which is written as a vector W3):

```
W3 = [16.11 % Total weight of Mg in the 3 samples
1.560 % Total weight of Al in the 3 samples
0.330]; % Total weight of Zn in the 3 samples
```

Create the vector W3 in your code.

Part 4 (3 pts) Let S = vector with the unknown weights of the 3 samples. Solve for S using the matrix inverse method.
Part 5 (3 pts) Use inverse or left matrix-division to solve for S using the left-division method.

Part 6. (2 pts) Multply M3 times S and verify that the result = W3

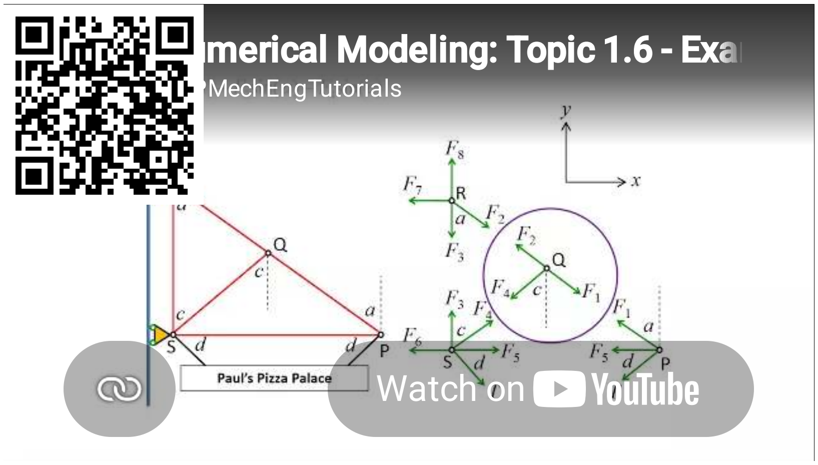
(https://www.azom.com/article.aspx?ArticleID=8675 (Links to an external site.))

Answer

The answer is not given here.

.

Truss problem



merical Modeling: Topic 1.6 - Exa
MechEngTutorials
y
x
F_8
F_7
R
F_2
a
F_3
F_2
Q
Q
c
F_4
c
F_1
F_3
F_4
F_1
c
a
a
c
F_6
a
S
d
d
P
S
d
F_5
F_5
d
P
Paul's Pizza Palace
Watch on YouTube

.

This page titled 13.5: Systems of Equations Examples and Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

13.6: Over-Determined Systems of Equations

13.6.1: Consistent Equations

Simple examples of consistent over-determined system of equations:

- One equation is a multiple of another equation
- One equation is the of 2 other equations

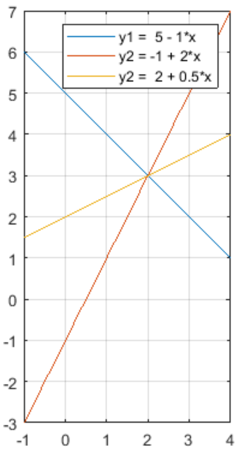
The rank of a consistent set of equations is equal to the number of unknowns.

Example 13.6.1 "Overdetermined" consistent system: 3 equations, 2 unknowns

x = -1 : 0.2 : 4;
y1 = 5 - 1*x;
y2 = -1 + 2*x;
y3 = 2 + 0.5*x;

The 3 lines intersect in 1 point, so these equations are consistent.

Graphical intersection at (2, 3)



%% Matrix reformulation:

% Put the variables on the left and the constants on the right.
% 1*x + y = 5
% -2*x + y = -1
% -0.5*x + y = 2

```
% A*xy = B
A = [ 1  1
     -2  1
    -0.5 1]
B = [ 5
     -1
      2]
rank(A) % 2
xy = A\B % Matrix algebra solution = (2, 3)
```

Solution

Add example text here.

13.6.2: .

Exercise 13.6.1 Over-Determined Consistent System of Equations 1

Over-Determined consistent system of equations of 4 equations in 3 unknowns.

```
A3 = [3  2  5
      4  5 -2
      1  1  1
      2 -4 -7]
% X3 = [x y z]'
B3 = [22
       8
       6
     -27]
```

Use Left matrix-division to solve this system of equations for X3

Check your solution by computing this:

```
X3ck = A3*X3
```

The result should be close to B3.

Answer

Add texts here. Do not delete this text first.

.

13.6.3: Inconsistent Equations

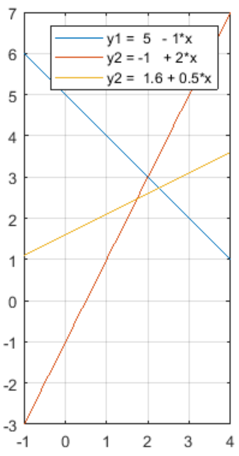
Example 13.6.2 "Overdetermined" inconsistent system: 3 equations, 2 unknowns

```
x = -1 : 0.2 : 4;
y1 = 5 - 1*x;
y2 = -1 + 2*x;
y3 = 1.6 + 0.5*x;
```

% The 3rd equation has been modified, so the 3 lines do not intersect in a single point.

Measurement Errors were added to the right-hand-side values of the consistent version of this exercise. Extra measure ments are often taken to help average-out measurement errors.

These these equations are inconsistent. Rather, there are 3 intersection points of each pair of lines.



%% We can still use left-division (Gaussian elimination)
% to find a good approximate solution.
% Matrix formulation
% 1*x + y = 5
% -2*x + y = -1
% -0.5*x + y = 1.6]
% xy = [x
% y]
% A*xy = B]

```
A = [ 1 1
-2 1
-0.5 1]
B = [ 5
-1
1.6]
rank(A) % 2
xy = A\B % Solution = (2, 2.87)
```

Solution

Add example text here.

.

Exercise 13.6.2 Over-Determined Inconsistent System of Equations 2

Over-Determined inconsistent system of equations of 4 equations in 3 unknowns.

Measurement Errors were added to the right-hand-side values of the consistent version of this exercise. Extra measure ments are often taken to help average-out measurement errors.

```
A4 = [3  2  5
      4  5 -2
      1  1  1
      2 -4 -7]
% X4 = [x y z]'
B4 = [22+1
       8-0.5
      6+1
    -27-1]
```

Use Left matrix-division to solve this system of equations for X4
Check your solution by computing this:
X4ck = A4*X4

The result should be somewhat close to B4, but will not match because these equations are inconsistent.

Answer

Add texts here. Do not delete this text first.

.

This page titled 13.6: Over-Determined Systems of Equations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

13.7: Chapter 13 Linear Algebra Videos in English and Espanol

English

roduction to Linear Algebra with
TLAB
Online Course
INTRODUCTION TO LINEAR ALGEBRA WITH MATLAB
With Ben Placek
Watch on YouTube

.

Espanol

ATLAB PARA ALGEBRA LINEAL 01
as
TLAB
CON ENFASIS EN ALGEBRA LINEAL
Watch on YouTube

ATLAB PARA ALGEBRA LINEAL 02
as
TLAB
CON ENFASIS EN ALGEBRA LINEAL
Watch on YouTube

.

This page titled 13.7: Chapter 13 Linear Algebra Videos in English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

14: Advanced Plotting

14.1: scatter, scatter3

14.2: Plot 3D Surface and Contour Plots

14.3: Advanced Plot Formatting

14.4: Log plots

14.5: Subplots

14.6: Chapter 14 Advanced Plotting Videos in English and Espanol

By Carey A. Smith

Octave needs a graphics toolkit to plot graph. Insert this line in your code before creating your first figure:

```
graphics_toolkit("fltk") % Do not use with MATLAB
```

For more information on graphics toolkit options, see:

https://docs.octave.org/interpreter/Graphics-Toolkits.html

14.1:

This page titled 14: Advanced Plotting is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

14.1: scatter, scatter3

14.1.1: The plot() and scatter() functions can make similar 2D plots

Example 14.1.1 2D plot() and scatter()

```
%% Create random x and y data points
n = 20;
x = rand(1, n); % a 1xn vector of random values between 0 and 1
y = rand(1, n); % another 1xn vector of random values between 0 and 1
```

Compare the results of these codes.

```
%% Plot with the plot function, open circles
figure;
plot(x, y,'o')
grid on
```

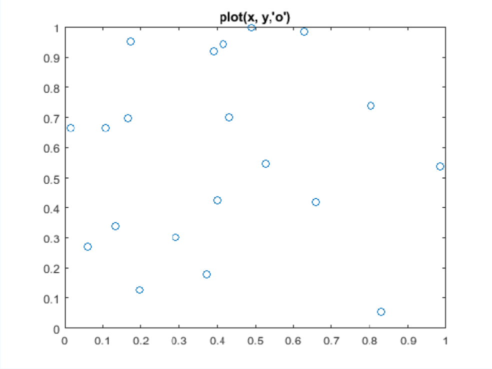


Figure 14.1.1: plot(x, y,'o')

```
%% Plot with the scatter function, open circles
figure;
scatter(x, y)
grid on
```

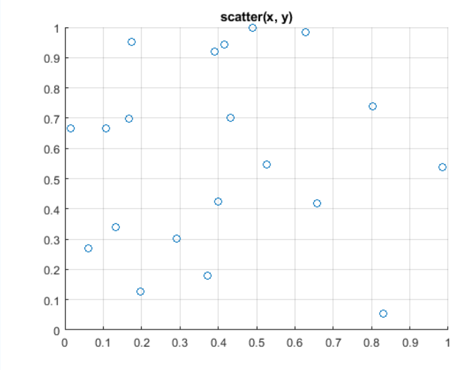
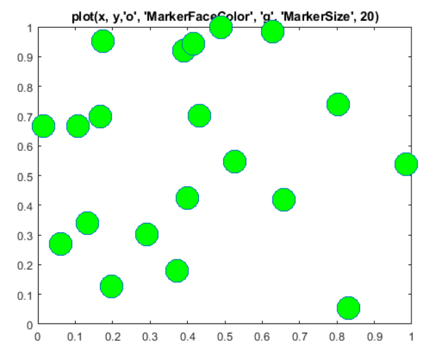


Figure 14.1.2: scatter(x, y)

The markers can be filled in and the size changed with both plot() and scatter(), but the options are specified differently:

```
%% Plot with the plot function, large, filled circles
figure;
plot(x, y, 'o', 'MarkerFaceColor', 'g', 'MarkerSize', 20)
title('plot(x, y,''o'', ''MarkerFaceColor'', ''g'', ''MarkerSize'', 20)')
```

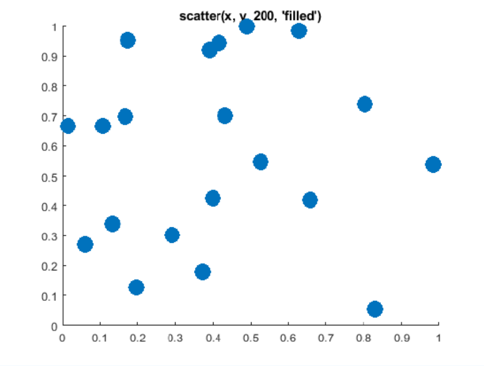
Figure 14.1.3: plot() with MarkerFaceColor and MarkerSize options



%% Plot with the scatter function, large, filled circles

```
figure;
scatter(x, y, 200, 'filled') % size = 20, filled markers
title('scatter(x, y, 200, ''filled'')')
```

Figure 14.1.4: scatter() with size and 'filled' options



Solution

Add example text here.

.

Example 14.1.2 xlim, ylim, text

This shows the use of the xlim() and ylim() functions with a 2D plot of a curve. It also shows how to use the text() function to add information to a figure. The code is attached in the file xlim_ylim_example.m

```
%% Example for xlim, ylim, axis
format short g; format compact; close all; clear all; clc;
x = -6:0.1:6;
y = -1 -5*x -0*x.^2 +0.3*x.^3;
figure;
plot(x,y, 'LineWidth',4);
grid on;
title('y = -1 -5*x -0*x^2 +0.3*x^3')

%% Zoom in on x
xlim([-3, 4]) % [x_lower_limit, x_upper_limit]

%% Zoom out in y
ylim([-1, 1]*12) % [y_lower_limit, y_upper_limit]

%% Look for the zero crossing by zooming in on both x & y in 1 command
axis([-1 1 -1 1]*0.5)
```

```
% [x_low, x_up, y_low, y_up, ]
% y = 0 at about x = -0.2

%% Check
y_len = length(x) % 121
y(51:60)
% W can see the the closest value to 0 is:
y(59) % -0.0024
% The corresponding x value is:
x(59) % -0.2
axis([-0.3 0.1 -.1 .1]) % zoom in

%% text() example
axis auto % reset the axes to their original scaling
axis('auto')
text(x(59), 0, ['a zero near ',num2str(x(59))])

%% We see that the text is ovelapping with the curve.
% So replot the curve on a new figure and raise the position of the text.
figure;
plot(x,y,'g','LineWidth',2);
grid on;
title('y = -1 -5*x -0*x.^2 +0.3*x.^3')
text(x(59), 2.0, ['a zero near x= ',num2str(x(59))])

%% Put texts near the other zeros
text(-4, 10, ['a zero near x= ',num2str(-4)])
text(4.2, -5, ['a zero near'])
text(4.2, -8, ['x= +4'])

%% Plot arrows pointing to the zeros
hold on;
quiver(-4,10, 0,-10, 'LineWidth',2)
%quiver(x,y, dx,dy)
quiver(4.2,-5, 0,5, 'LineWidth',2)
%quiver(x,y, dx,dy)
```

Solution

Add example text here.

.

? Exercise 14.1.1 xlim, ylim assignment

A. Open a new figure and plot these 2 functions on it:

```
x = (0:0.05:1)*pi;
y1 = 0.3*x;
y2 = cos(2*x);
figure(1)
plot(x,y1)
hold on;
plot(x,y2)
grid on;
```

We can see that these functions intersect near x = 0.7 and y = 0.2

B. Use xlim, ylim to zoom in on the area near (0.7, 0.2).
Set the x-limits to be [0.5, 1.0]
Set the y-limits to be [0.1, 0.3]

Save the figure and submit it.

The figure can be saved in 2 ways:

1. Under the File menu, select "Save as" and file type *.png or *.jpg and give it a file name.

2. Under the Edit menu, select "Copy Figure", then paste it into a file. This is convenient when writing a report with multiple figures.

Answer

The answer is not given here.

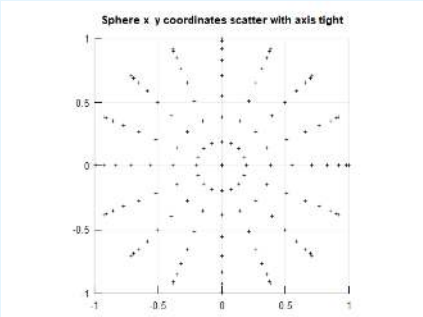
.

✓ Example 14.1.3 Sphere x & y coordinates scatter 2D with axis tight

```
[x, y, z] = sphere(16); % 17 x 17 grid
x_size = size(x)
% For the scatter functions, the x, y, and z data need to be 1D
x1 = x(:);
y1 = y(:);
z1 = z(:);
x1_size = size(x1)
%% B. Open a new figure
figure
% Use the scatter() function to make
%  a 2D scatter plot of the x & y coordinates
scatter( x, y, 'k','+');
% In Octave, the marker symbol and color are specified separately.
grid on
axis equal % Set the x & y scales to be the same
axis tight % Bring in the x and y axis limits
```

Solution

Figure 14.1.5: Sphere scatter 2D



.

✓ Example 14.1.4 3D sphere with scatter3()

```
%% scatter_plots_sphere.m
format compact; format short; clear all; clc; close all;
% scatter3() can plot data that is not on a rectangular grid.

%% This example is adapted from Matlab's scatter help page.
% sphere() is a built-in example
[X,Y,Z] = sphere(16); % 17 x 17 grid
X1D = X(:); % Convert X from a 17x17 array to a 289x1 vector
Y1D = Y(:); % Convert Y from a 17x17 array to a 289x1 vector
Z1D = Z(:); % Convert Z from a 17x17 array to a 289x1 vector

%% Use scatter3() to create 3D plots.
figure
scatter3(X1D, Y1D, Z1D,'go','filled')
% 'go' = green circles for the data markers
% 'filled' = fills-in the circles
title('3D scatter plot example: sphere')
xlabel('X')
ylabel('Y')
zlabel('Z')
axis equal

%% The view angle can be changed
[az, el] = view % This is the current view angle
% az = -37.5000
% el = 30
view(45,10) % view(azimuth, elevation)

%% Try adjusting the axis limits by evaluating each of these lines, 1 at a time.
xlim([-2,2])
ylim([0,1])
zlim([0,1])
xlim([-1,1])
view(70,30)
view(10,10)
```

Solution

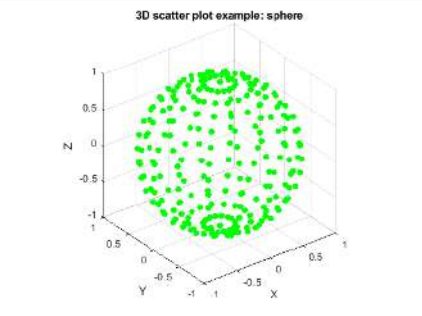


Figure 14.1.6: Sphere scatter3 defaut view

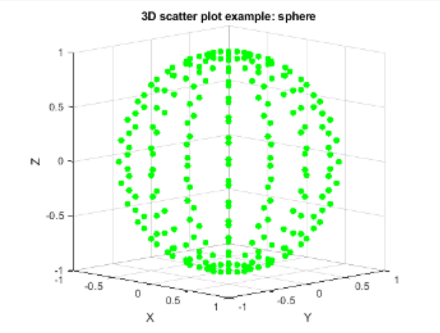


Figure 14.1.7: Sphere scatter3 alternate view

.

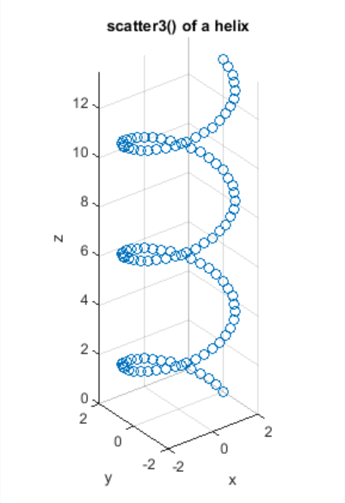
Next we make a 3-dimensional scatter3() plot.

Example 14.1.5 scatter3() helix

```
%% scatter3D_helix.m
format short g; format compact; clear all; close all; clc;
%% The scatter3() function makes a 3D plot of individual points
% The inputs to the scatter3() function are x, y, and z vectors defining each point.
% Unlike other 3D plotting functions, it does not use a 2D grid of x and y
coordinates

% Define these x and y vectors:
r = 2; % radius of the helix
ang = 0 : 10 : 3*360; % angles (degs)
x = r*cosd(ang);
y = r*sind(ang);
n = length(ang);
z = (0 : (n-1)) /8;

%% Open a figure and plot this surface using the scatter3() function
% scatter3 expects the 3 inputs to be vectors.
figure;
scatter3(x, y, z);
axis equal

%%
title('scatter3() of a helix');
xlabel('x') % This identifies x-axis
ylabel('y')
zlabel('z')
% You can rotate the graph in 3D.
% You can zoom in.
```

Solution

Figure 14.1.8: scatter3 helix



Exercise 14.1.2 scatter3 for an undersea mountain

The file scatter3_seamount_start.m code loads the seamount.mat data file. The seamount.mat file is attached to this section. You need to download it. The scatter3_seamount_start.m file and the seamount.mat both need to be in the same directory as the command window.

You need to add the scatter() function to plot the x, y coordinates and the scatter3() function to plot the full 3D seamount. Change the name of the completed .m file scatter3_seamount_YourName.m

Answer

Your figure should look like this:

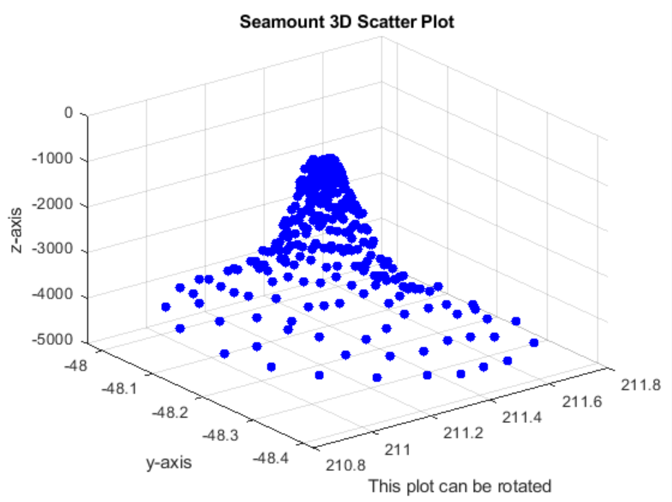


Figure 14.1.9: Sphere scatter3 seamount

.

This page titled 14.1: scatter, scatter3 is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

14.2: Plot 3D Surface and Contour Plots

By Carey A. Smith

14.2.1: meshgrid() function

Surface plots need 2-dimensional grids of x and y coordinates. These are made from 1-dimensional vectors of x and y coordinates using the meshgrid() function. The syntax of meshgrid() is:

[x2D, y2D] = meshgrid(x1D,y1D);

✓ Example 14.2.1 Simple meshgrid() Example

x1D = [0, 1, 2 ,3];
y1D = [0, 2, 4];
[x2D, y2D] = meshgrid(x1D,y1D)

Solution

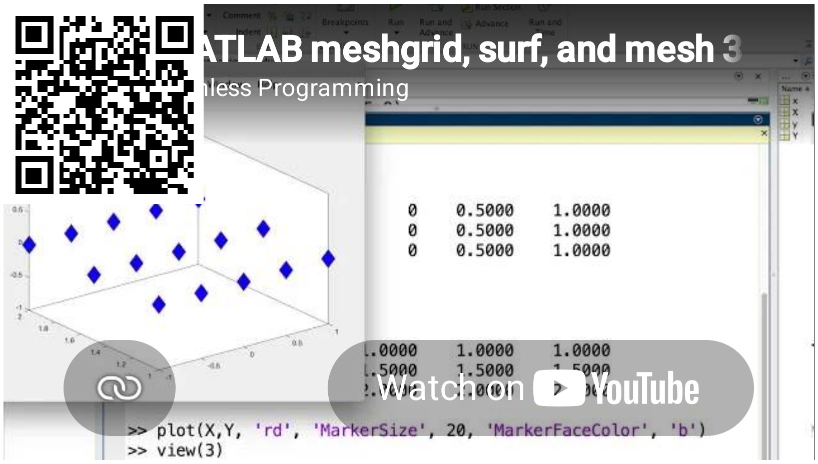
x2D =
0 1 2 3
0 1 2 3
0 1 2 3
y2D =
0 0 0 0
2 2 2 2
4 4 4 4

As can be seen:

The x coordinates are repeated in all the rows.

The x coordinates are repeated in all the columns.

Watch this video about meshgrid. Note that the presenter uses lower case x & y for 1-dimensional values and upper case X & Y for 1-dimensional values. It is not good practice to use variables names that only differ by lower and upper case. Better is to use different name, like x1D, y1D and x2D, y2D. The important part is 0 to 4:20 minutes.



```
     0    0.5000    1.0000
     0    0.5000    1.0000
     0    0.5000    1.0000

>> plot(X,Y, 'rd', 'MarkerSize', 20, 'MarkerFaceColor', 'b')
>> view(3)
```

.

14.2.2: mesh() 3D Plot

The mesh() function creates a wire-frame 3-dimensional plot.

✓ Example 14.2.2 mesh() 3D Plot

```
%% Create a 3D graph of r^2 = x^2 + y^2 over a grid of points
% 1. Set the x and y axis 1D vectors that define this rectangular grid:
x1D = -3 : 0.5 :3; % 1x13 vector
y1D = -2.5: 0.5 :2.5; % 1x11 vector

%% 2. Matlab's meshgrid() function uses these 1D vectors
% to create 2D arrays of x & y at each point of the rectangular grid.
[x2D, y2D] = meshgrid(x1D,y1D);

%% 3. Compute z(x,y) at each point in the grid.
z = x2D.^2 + 1*x2D + y2D.^2 + 2*y2D;

%% mesh(x,y,z) creates a 3D wire frame plot.
figure;
mesh(x2D, y2D, z);
title('mesh(x2D, y2D, Z)');
colormap jet % This selects a colormap
colorbar; % This displays the color scale
xlabel('x') % This identifies x-axis
ylabel('y')
zlabel('z')
```

Solution

The resulting plot is shown here.

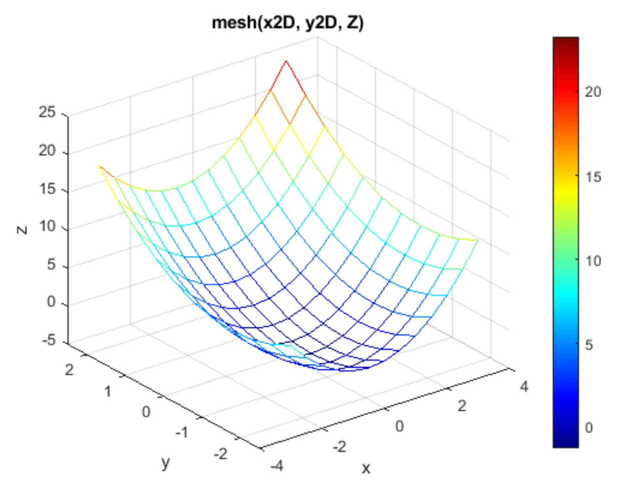


Figure 14.2.1: mesh() 3D plot

.

14.2.3: surf() 3D plot

The surf() function also creates a continous surface 3-dimensional plot. Its usage is the same as the mesh() function.

✓ Example 14.2.3 surf() 3D Plot

```
format short g; format compact; clear all; close all; clc;

%% Create a 3D graph of r^2 = x^2 + y^2 over a grid of points
% 1. Set the x and y axis 1D vectors that define this rectangular grid:
x1D = -3 : 0.5 :3; % 1x13 vector
y1D = -2.5: 0.5 :2.5; % 1x11 vector
```

```
%% 2. Matlab's meshgrid() function uses these 1D vectors
% to create 2D arrays of x & y at each point of the rectangular grid.
[x2D, y2D] = meshgrid(x1D,y1D);

%% 3. Compute z(x,y) at each point in the grid.
z = x2D.^2 + 1*x2D + y2D.^2 + 2*y2D;

%% surf(x,y,z) creates a 3D continous surface plot.
figure;
surf(x2D, y2D, z);
title('mesh(x2D, y2D, Z)');
colormap hot % This selects a colormap
colorbar; % This displays the color scale
xlabel('x') % This identifies x-axis
ylabel('y')
zlabel('z')
```

Solution

The resulting plot is shown here.

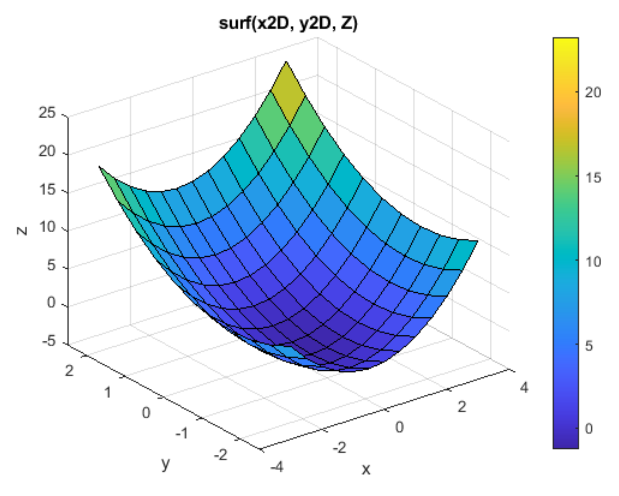


Figure 14.2.2: surf() 3D plot

See this attached file for images of the colormap options.: colormap_options.pdf

.

14.2.4: contour() Plot

Contour plots are like topographical maps. They display 3D data in a 2D format. The basic syntax is:

contour(x2D, y2D, z);

This can be improved by specifying the contour levels and showing their values:

contour(x2D, y2D, z, levels, 'ShowText','on')); % This creates specified contours and shows their values.

✓ Example 14.2.4 contour() Plot

```
format short g; format compact; clear all; close all; clc;

%% Create a 3D graph of r^2 = x^2 + y^2 over a grid of points
% 1. Set the x and y axis 1D vectors that define this rectangular grid:
x1D = -3 : 0.5 :3; % 1x13 vector
y1D = -2.5: 0.5 :2.5; % 1x11 vector

%% 2. Matlab's meshgrid() function uses these 1D vectors
% to create 2D arrays of x & y at each point of the rectangular grid.
[x2D, y2D] = meshgrid(x1D,y1D);
```

```
%% 3. Compute z(x,y) at each point in the grid.
z = x2D.^2 + 1*x2D + y2D.^2 + 2*y2D;

%% contour(x,y,z) creates a continous surface plot.
figure;
levels = [-4: 2: 24]; % Specify the contours
contour(x2D, y2D, z, levels, 'ShowText','on');
title('contour(x2D, y2D, Z), levels, ''ShowText'',''on'')');
colorbar; % This displays the color scale
```

Solution

Resulting plot:

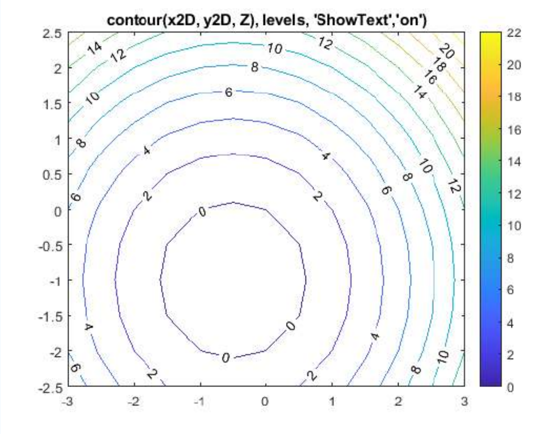
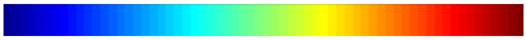
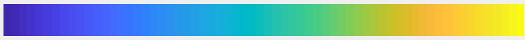
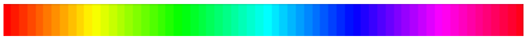
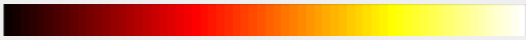
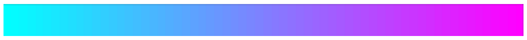
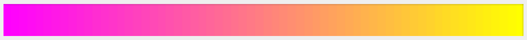
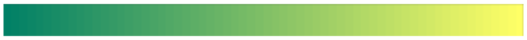
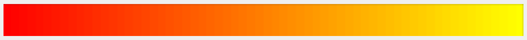
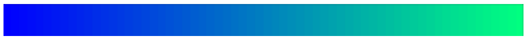
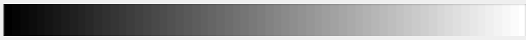
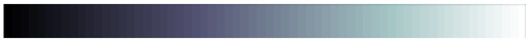
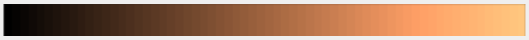
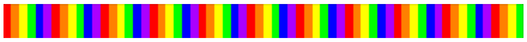
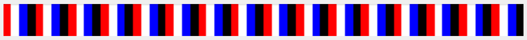
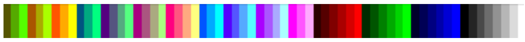
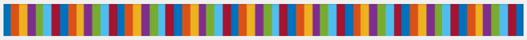


Figure 14.2.3: contour() plot

.

.

14.2.5: colormap Options

jet	
parula	
hsv	
hot	
cool	
spring	
summer	
autumn	
winter	
gray	
bone	
copper	
prism	
flag	
colorcube	
lines	

This page titled 14.2: Plot 3D Surface and Contour Plots is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

14.3: Advanced Plot Formatting

By Adam L. Lambert, chapter 15

In this section we will take a look at some more detailed plot formatting. For readability, the code presented will focus on the concept being discussed. Replicating example images may require additional code. For example, it is assumed that the basics of title, legend, axis, etc. are understood. These commands will only be shown when necessary. Also, initial examples will show the calculation of the data to be plotted. As we move through the chapter this code will be omitted for brevity.

Some of this formatting will utilize options which can be passed directly into a function, others will be accesses through an object. These concepts will be discussed in the examples.

Line and Marker Size

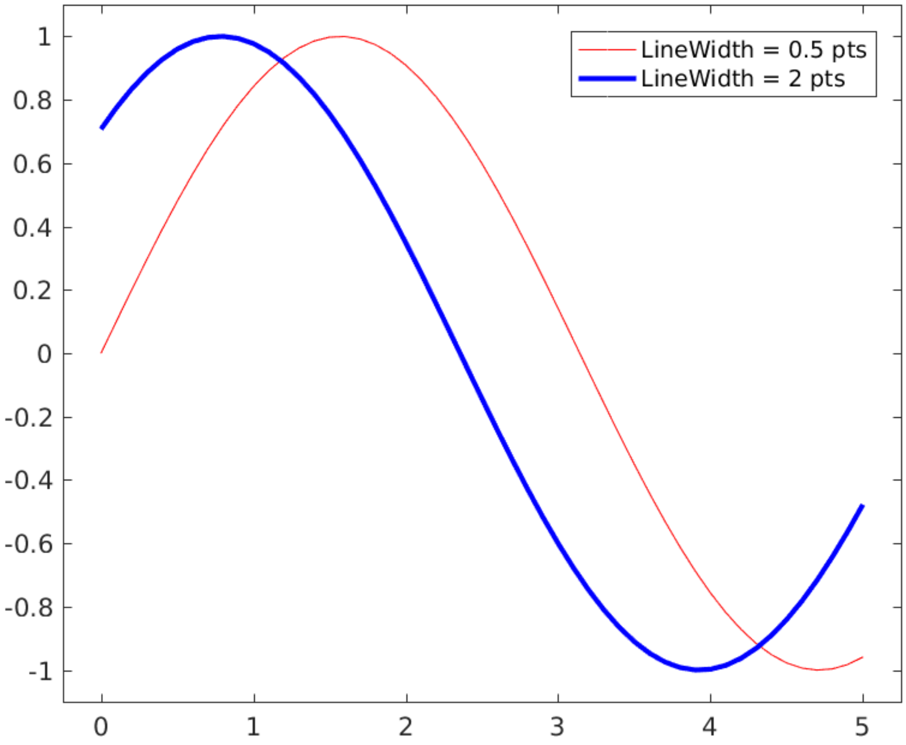
MATLAB allows us the specify the thickness of a line or the size of a marker. The unit for each of theses pecifications is called a point. A point is equal to 1/72 of an inch and is the smallest unit of measure in typography.

LineWidth

The width of a line can be specified using the LineWidth option for the plot command. Like many other options in MATLAB, this option is defined using a name-value pair. We can pass the name of the option and the value that we want the option to have directly into the plot command. Here is a short example plotting two lines with different thicknesses. When we plot y2, the LineWidth option is passed into the plot command. Note that the option is in string quotes. The value that follows is the value that we will set for the option.

```
clear all, format compact, close all;

% x-array
x = [0:0.1:5];
% y-arrays
y1 = sin(x);
y2 = sin(x+(pi/4));

figure
% plot y1 with default width (0.5 pts)
plot(x,y1,'r-')
hold on;
% plot y1 with custom width (2 pts)
plot(x,y2,'b-', 'LineWidth', 2)
```

As you can see in the plot below, the blue line is signicantly thicker than the red line. Using bold lines often makes plots easier to read and is advised in most cases. The exact size will have the be determined on a case by case basis, but each plot should be easy to look at and interpret.

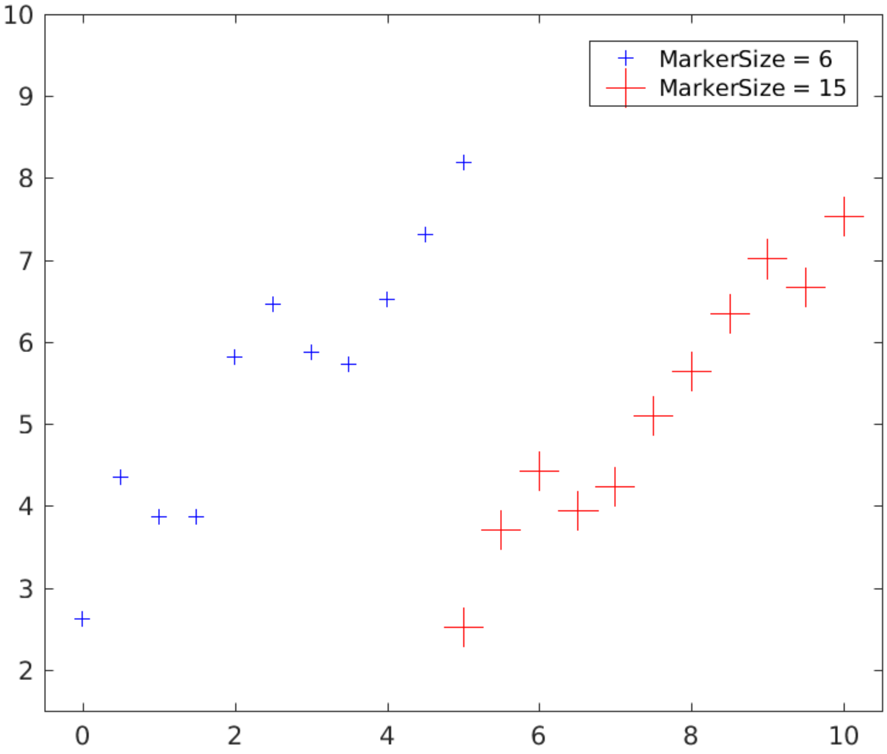


Caption: Results of the LineWidth option.

MarkerSize

For scatter plots of discrete data, the MarkerSize option is used to change the size of the marker. This option is a name-value pair as well. The default value for this option is 6 points. The code below compares the default size of the + marker with a custom size of 15 points.

```
n = 11;
% data 1
x1 = linspace(0,5,n);
y1 = linspace(2,7,n) + 2*rand(1,n);
% data 1
x2 = linspace(5,10,n);
y2 = linspace(2,7,n) + 2*rand(1,n);

figure
% plot y with default marker size
plot(x1,y1,'b+')
hold on;
% plot y with custom marker size
plot(x2,y2,'r+', 'MarkerSize', 15)
```



Caption: Results of the MarkerSize option.

Line and Marker Color

In earlier chapters we used letters to represent the colors that we were plotting. The letters were passed in as part of the LineSpec and we used 'r' for red and 'b' for blue, etc. In this chapter we'll learn how to specify colors using an RGB triplet. The colors created by your monitor, or any LED light, are all combinations of red, green, and blue. You can read about the RGB color model on Wikipedia.

https://en.Wikipedia.org/wiki/RGB_color_model

By passing an array with a value for red, a value for blue, and a value for green into the plot command, we can create custom colors for our plots.
Often the values for each channel range from 0-255 where 0 is completely absent and 255 is fully saturated. This is because there are 256 possibilities for an 8-bit slot of memory. MATLAB requires the values to be between 0 and 1, so we can look up RGB values for colors online and then divide them by 255.

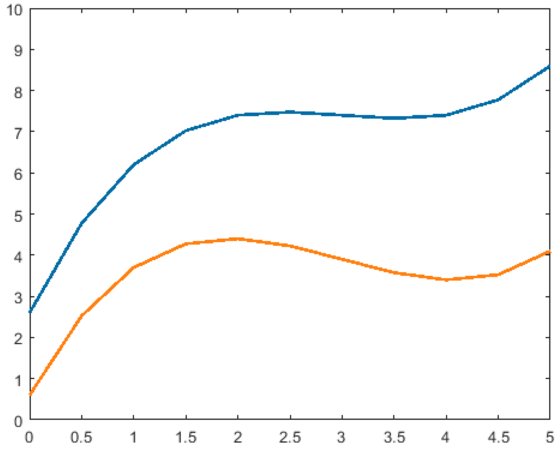
Lines

This code will create two custom colors and then plots some lines. Also the line width is customized.

```
x1 = 0 : 0.5 : 5;
y1 = 8 - 0.2*x1 + 0*x1.^2 + 0.2*(x1-3).^3;
y2 = 6 - 0.7*x1 + 0*x1.^2 + 0.2*(x1-3).^3;
% define custom colors
custom_blue = [0,107,164]./255;
custom_orange = [255,128,14]./255;
figure;
% plot data with custom colors
plot(x1,y1,'-', 'Color', custom_blue, 'LineWidth', 2)
```

```
hold on;
plot(x1,y2,'-', 'Color', custom_orange, 'LineWidth', 2)
ylim([0,10])
```

As you can see, we use another name-value pair for the color option. In this case the value is an array of RGB values scales from 0-1. The first element is the red channel, the second element is the green channel, the third is the blue channel.



Caption: Lines plotted with custom colors.

As you can see these colors are softer and more aesthetic than the fully saturated values we get from the LineSpec.

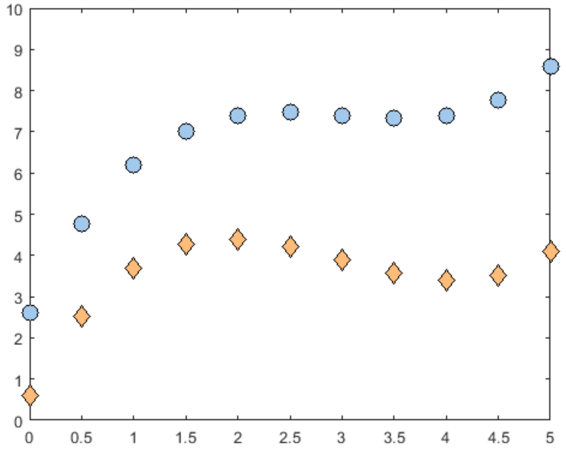
Markers

Markers actually have two colors which can be changed, the border and the fill. This example will plot markers with a custom color for the fill, and the default black for the border. The size is also changed. Because the name-value pairs of the options are so long, we can split the command into multiple lines using the ellipsis (...) to break up the lines.

```
close all
% Lambert's Figure 15.3 doesn't specify the x1 & y1 vectors
% This is an approximation of y1 and y2
x1 = 0 : 0.5 : 5;
y1 = 8 - 0.2*x1 + 0*x1.^2 + 0.2*(x1-3).^3;
y2 = 6 - 0.7*x1 + 0*x1.^2 + 0.2*(x1-3).^3;
```

```
% define custom colors
custom_light_blue = [162,200,236]./255;
custom_light_orange = [255,188,121]./255;
figure;
% plot y with custom marker color
plot(x1,y1,'o','MarkerEdgeColor','k'...
,'MarkerFaceColor',custom_light_blue...
,'MarkerSize', 10)
hold on;
% plot y with custom marker color
plot(x1,y2,'d','MarkerEdgeColor','k'...
,'MarkerFaceColor',custom_light_orange...
,'MarkerSize', 10)
ylim([0,10])
```

The results of this code will look like this.



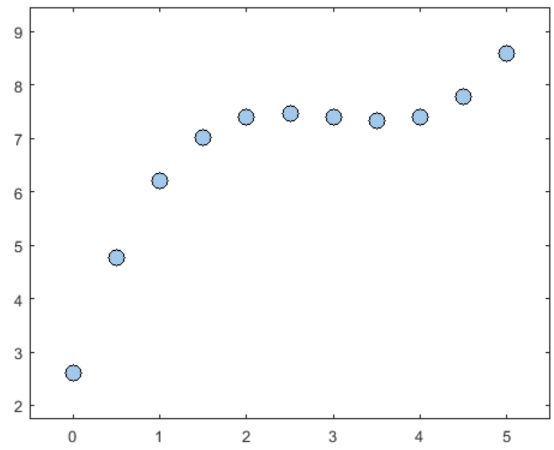
Caption: Markers with custom colored fill and black borders.

Axis

The axis command is fairly straightforward, but we can speed things up by automatically calculating the upper and lower bounds for each axis. This allows a plotting script to adapt to new data sets without manual editing. In this example we calculate the upper and lower bounds by finding the minimum and maximum values for each array and then subtracting a value which is scaled relative to the maximum value.

```
close all
x1 = 0 : 0.5 : 5;
y1 = 8 - 0.2*x1 + 0*x1.^2 + 0.2*(x1-3).^3;
% define custom colors
custom_light_blue = [162,200,236]./255;
figure;
% plot y with custom marker color
plot(x1,y1,'o','MarkerEdgeColor','k'...
,'MarkerFaceColor',custom_light_blue...
,'MarkerSize', 10)
% find x-axis limits
x_scale = 0.1;
x_min = min(x1) - x_scale*max(x1)
x_max = max(x1) + x_scale*max(x1)
% find y-axis limits
y_scale = 0.1;
y_min = min(y1) - y_scale*max(y1)
y_max = max(y1) + y_scale*max(y1)
% pass limits to axis command
axis([x_min, x_max, y_min, y_max])
```

In this case we use a scaling factor of 0.1 so the buffer will be 10% of the maximum value for each array. We add it to the upper limit and subtract it from the lower limit. You can use whatever scaling factor makes your plot look nice.



Caption: Automated axis scaling.

.

This page titled 14.3: Advanced Plot Formatting is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

14.4: Log plots

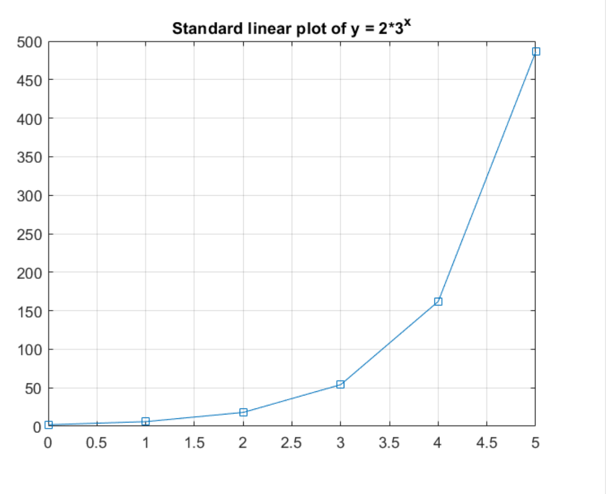
When plot values cover more than 1 order of magnitude, it can be more useful to create a logarithm plot rather than a standard lineat plot.

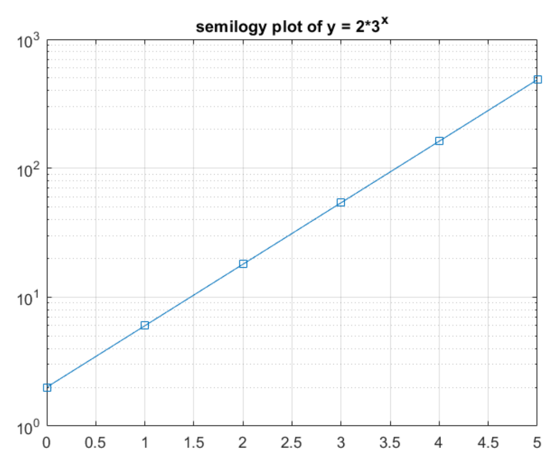
The MATLAB and Octave functions for logarithm plots are semilogx(), semilogy(), and loglog(). The following examples illustrate the use of these functions.

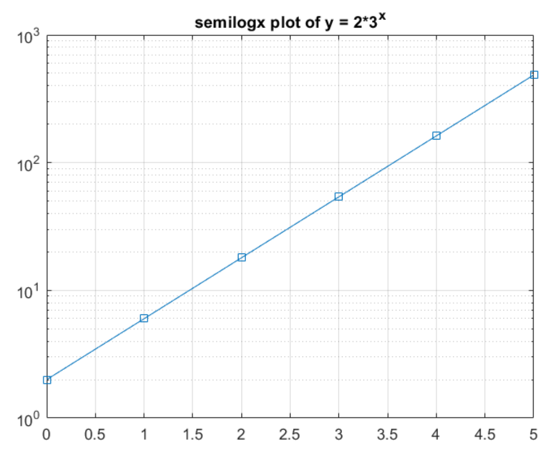
✓ Example 14.4.1 semilogy(), log(), semilogx()

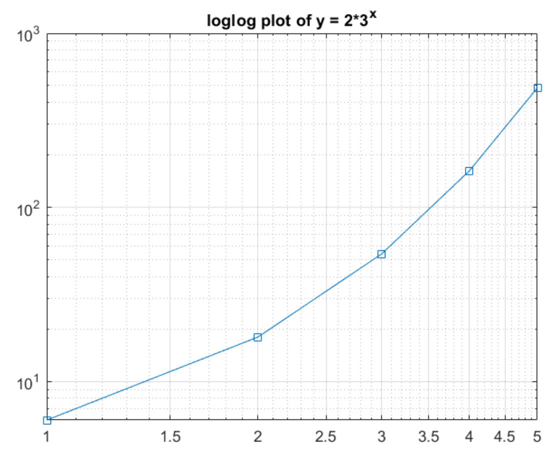
```
% Log plot examples

% Create data
x = 0:5
y = 2*3.^x

%% Standard linear plot
figure;
plot(x,y,'-s')
title('Standard linear plot of y = 2*3.^x')
grid on

%% semilogy plot
figure;
semilogy(x,y,'-s')
title('semilogy plot of y = 2*3.^x')
grid on

%% semilogx plot
figure;
semilogx(x,y,'-s')
title('semilogx plot of y = 2*3.^x')
grid on
%% loglog plot
figure;
loglog(x,y,'-s')
title('semilogy plot of y = 2*3.^x')
grid on
```

Solution









Add example text here.

.

This page titled 14.4: Log plots is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

14.5: Subplots

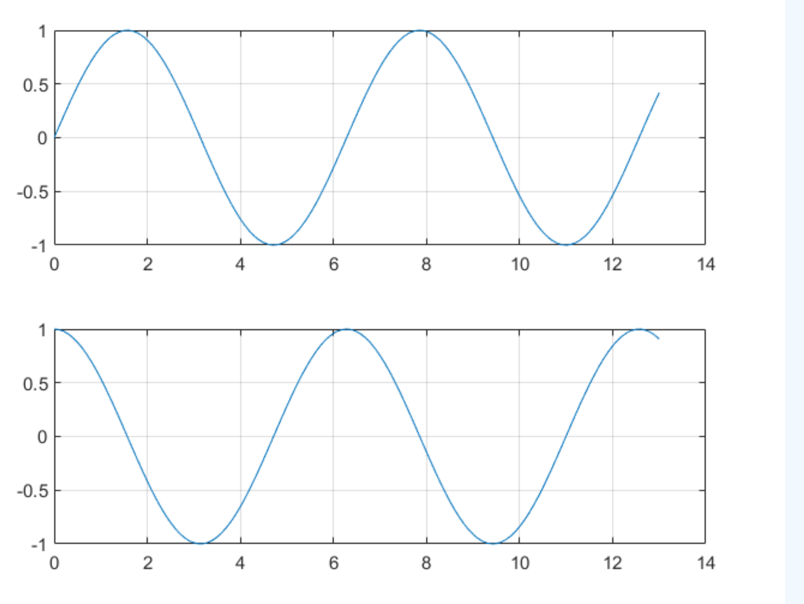
The subplot() function allows you to place more than 1 plot within a single figure.

These examples show its use. More information can be obtained by using "help subplot"

✓ Example 14.5.1 subplot() with 2 rows & 1 column

```
%% 2 Subplots
figure
subplot(2,1,1);
x = 0 : 0.1 : 13;
y1 = sin(x);
plot(x,y1)
grid on;

subplot(2,1,2);
y2 = cos(x);
plot(x,y2)
grid on;
```

Solution



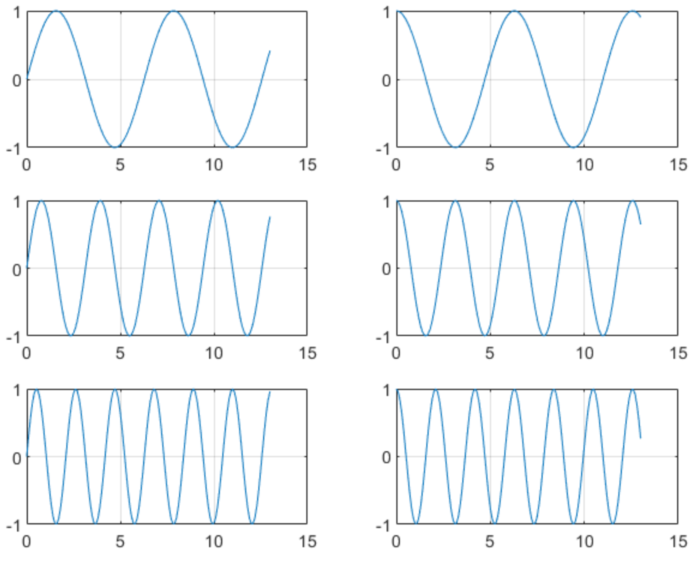
.

✓ Example 14.5.2 subplot() with 3 rows & 2 columns

```
%% 3x2 Subplots
figure
subplot(3,2,1);
x = 0 : 0.1 : 13;
ys1 = sin(x);
plot(x,ys1)
grid on;
```

```
subplot(3,2,3);
x = 0 : 0.1 : 13;
ys2 = sin(2*x);
plot(x,ys2)
grid on;

subplot(3,2,5);
x = 0 : 0.1 : 13;
ys3 = sin(3*x);
plot(x,ys3)
grid on;

subplot(3,2,2);
yc1 = cos(x);
plot(x,yc1)
grid on;

subplot(3,2,4);
yc2 = cos(2*x);
plot(x,yc2)
grid on;

subplot(3,2,6);
yc3 = cos(3*x);
plot(x,yc3)
grid on;
```

Solution



.

This page titled 14.5: Subplots is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

14.6: Chapter 14 Advanced Plotting Videos in English and Espanol

14.6.1: English

ATLAB: Surface Plot Examples
on Feed Me
rface Plot
Examples
MATLAB
Watch on YouTube

Plotting in Matlab
stopher Lum
Watch on YouTube

.

14.6.2: Espanol

PLICACION GRAFICOS 3D en Mat
eniia
CONTOUR3
SURF
MESH
MATLAB
PLOT3
GRAFICOS 3D EN MATLAB
Watch on YouTube

últiples gráficas - 21 - Octave
siored
Octave
21 Múltiples Gráficas
Watch on YouTube

This page titled 14.6: Chapter 14 Advanced Plotting Videos in English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

15: Formatted Input and Output

15.1: sprintf() and fprintf(), Introduction
15.2: fprintf() to a file
15.3: Reading from an Excel file in MATLAB and Octave
15.4: Chapter 15 Videos English and Espanol

15: Formatted Input and Output is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

15.1: sprintf() and fprintf(), Introduction

This section describes how to use the sprintf() function to write to a string and how to use the fprintf() function to write to the screen.

The next section describes how to use the fprintf() function to print formatted messages and numbers to a file.

sprintf() and fprintf() for strings without numbers

sprintf means string print formatted. What this means, is that it can create a formatted message and store it in a character string variable. It could be used to create a title or part of a title to pass to a routine that makes plots.

title_str1 = sprintf('MATLAB Homework Plot')

fprintf means file print formatted. What this means, is that it can create a formatted message and store it in a file. When no file is specified, it prints to the screen, like this:

fprintf('Hello World\n')

The \n puts the cursor at the beginning of a new line. Otherwise, the cursor is at the end of the message just printed. A \n is needed for most fprintf() messages.

sprintf() and fprintf() with scalar numbers, simple formatting

To create a message with numbers, the numbers need to be converted to strings using one of these format strings:

Format string options:
%f = Fixed point notation (decimal)
%e = Exponential notation
%d = Decimal, w/out trailing zeros
%g = General (%f or %e, whichever is shorter)
%c = Character (one at a time)
%s = String

If you want to print the percent sign, you need to use a double percent: %%

Try these simple examples:

nbirds = 5;

str1 = sprintf('There are %d birds in the park.', nbirds)

fprintf('There are %d birds in the park.\n', nbirds)

You can output more than one variable in a single line of code. Try this example:

ncrows = 2;

nrobins = 4;

nseagulls = 8;

str2 = sprintf('There are %d crows, %d robins, and %d sea gulls in the park.\n', ncrows, nrobins, nseagulls)

fprintf('There are %d crows, %d robins, and %d sea gulls in the park.\n', ncrows, nrobins, nseagulls)

Note that in each case, there is just 1 message string which has 3 format codes.

sprintf and fprintf() are functions with multiple inputs.

The first of input of sprintf is the message character string, including any format codes. This is followed by any number inputs.

When printing to the screen, the first input of fprintf is also the message character string, including any format codes. This is followed by any number inputs.

In the following, the uses of fprintf() can be changed to sprintf() by adding a variable to store the result.

Example 15.1.1 fprintf of 1000*pi with multiple formats

Run these lines of code:

```
fprintf('%%f format: 1000*pi = %f \n', 1000*pi)
fprintf('%%e format: 1000*pi = %e \n', 1000*pi)
fprintf('%%g format: 1000*pi = %g \n', 1000*pi)
```

They should produce the output lines shown in the solution:

Solution

%f format: 1000*pi = 3141.592654
%e format: 1000*pi = 3.141593e+03
%g format: 1000*pi = 3141.59

.

Exercise 15.1.1 fprintf() of the speed of light

Write code similar to the above example that prints the speed of light with the same 3 formats as above.

Start by setting this variable:

```
c = 2.998e8 % Speed of light in a vacuum
```

The first fprintf statement should be:

```
fprintf('%%f format: c = %f \n', c)
```

Also, print c with the %e and % g formats, as in the example above.

Answer

The answer is not given here.

.

A single fprintf() statement can print more than one variable. It can also have a combination of words and numbers, as shown in this example:

Example 15.1.2 fprintf() with both words and numbers

```
% Example fprintf() with both words and numbers
% https://en.wikipedia.org/wiki/Denise_Curry#UCLA_statistics
name1 = 'Denise'
name2 = 'Curry'
points = 3198
FG_pct = 60.7 %
fprintf('UCLA Star %s %s scored %4.1f%% of her shots with a 4-year total of %d
points\n', ...
         name1, name2, FG_pct, points)
```

Solution

Output:

UCLA Star Denise Curry scored 60.7% of her shots with a 4-year total of 3198 points

.

? Exercise 15.1.2 fprintf() mean and standard deviation, simple formatting

Compute these variables:

data1 = 3*(rand(1,10) - 0.5);
data1_mean = mean(data1);
data1_std = std(data1);

A. Use fprintf, a %f format string, and the computed value of data1_mean to print a message like this:
data1 mean = 0.531842, where the printed number will be your value of data1_mean.

B. Use fprintf, 2 %f format strings, and the computed values of data1_mean and data1_std to print a message like this:

```
data1 mean = 0.531842, data1_std= 1.027824
```

where the printed numbers will be your values of data1_mean and data1_std.

Answer

The answer is not given here.

.

sprintf() and fprintf() specifying the number of digits

The format strings can also specify the number of total digits and decimal places.

format string %8.2f means: 8 total characters, 2 decimal places, floating point format

The 8 total characters include a negative sign, if appropriate, and the decimal point, in addition to the digits before and after the decimal point. For numbers in scientific notation, it also needs to include characters for the "e", and exponent sign, and 2 exponent digits.

The total characters value should be at least the number of decimal places + 3 for the decimal point, a negative sign, and at least one leading digit.

✓ Example 15.1.2 Specify total number of characters and decimal places

Try these lines of code:

```
fprintf(' %%8.2f format: pi =%8.2f \n', pi)
fprintf('%%11.8f format: pi =%11.8f \n', pi)
fprintf(' %%9.2g format: pi =%9.2g \n', pi)
fprintf(' %%9.2e format: pi =%9.2e \n', pi)
fprintf('%%10.2e format: pi =%10.2e \n', pi)
```

Solution

```
%8.2f format: pi = 3.14
%11.8f format: pi = 3.14159265
%9.2g format: pi = 3.1
%9.2e format: pi = 3.14e+00
%10.2e format: pi = 3.14e+00
```

.

? Exercise 15.1.3 fprintf() mean and standard deviation, specify digits

Compute these variables:

data1 = 10*randn(1,10)
data1_mean = mean(data1);
data1_std = std(data1);

Use fprintf to print to the screen the computed value of data1_mean with 6 total characters and 3 decimal places, and the computed value of data1_std with 5 total characters and 2 decimal places. The result should look similar to this, but with your values:

```
data1 mean = 0.532, data1_std= 1.03
```

Answer

Hint: Your format string should include these 2 formats: %6.3f and %6.2f

.

? Exercise 15.1.4 Mimic dice rolls

```
x = randi(6,1,36); % Create a vector with uniform random integers between 0 and 6.
```

2. Compute the mean and standard deviation of the vector.

3. Display the mean and standard deviation values to the screen using fprint( ) with a %f format code (with 2 decimal places) to create a string like:

“The mean= x.xx, and the std. dev.= y.yy”

where x.xx and y.yy will be replaced with your values.

Hint: a number like 1.34 needs 4 characters, so use %4.2f

You need to use 2 proper %f format strings. No credit if you you just type in the values.

4. Make a similar output using disp( ) and num2str( ) functions with a single disp() function.

No credit if you you just type in the values.

This video explains how to display a number with the disp() command:

Creating formatted outputs using the disp() command in MATLAB. (Links to an external site.)

Hint: num2str() lets you specify the number of significant digits, as opposed to the number of decimal places. Since both the mean and the standard deviation are between 1 and 10, we want 3 significant digits to get 2 decimal places. So, you can use num2str(std_x, 3)

Answer

The answer is not given here.

.

fprintf() with a Matrix

One way to print the values of a matrix is to use nested for loops and print one value at a time. With this approach, you would only use the \n for the last element of a row.

However, fprintf() can print all of the values with a single instruction. If your message has a single format string, they will be printed as a column. If you have more than one format string, it will print as many values as there are format strings, then cycle back to the beginning of the format string to continue printing the remaining characters. This can result in printing the transpose of a square matrix. Consider this example of a square matrix.:

```
S = [1 2 3;
4 5 6;
7 8 9];
```

% Since there are 3 elements in each row, we try 3 %d format strings.

fprintf('%d %d %d\n',S)
% Result:
% 1 4 7
% 2 5 8
% 3 6 9
We see that it read the 3 values in the first column, but printed them in the first row. The printed matrix is the transpose of matrix S.

% This can be fixed by putting the transpose, S' in the fprintf statement:
fprintf('%d %d %d\n',S')
% Result is correct:
% 1 2 3
% 4 5 6
% 7 8 9

Consider this 2x4 matrix:

```
A = [ 1 2 3 4
5 6 7 8]
```

% Since there are 4 elements in each row, we try 4 %d format strings.

fprintf('%d %d %d %d\n', A)

% Result:
% 1 5 2 6
% 3 7 4 8

We see that it read the 2 values in the first column and then the 2 values in the second column and printed them in the first row. This is scrambled.
% This can be fixed by putting the transpose, A' in the fprintf statement:
fprintf('%d %d %d %d\n', A')
% Result:
% 1 2 3 4
% 5 6 7 8
.

? Exercise 15.1.5 fprintf 4x4 magic square

M4 = magic(4)
% 16 2 3 13
% 5 11 10 8
% 9 7 6 12
% 4 14 15 1
% Use fprintf() to print this matrix to the screen. Use 4 %d format strings and \n (newline) in the message argument. Use M4' for the numerical data.

Answer

Submit your code.

.

This page titled 15.1: sprintf() and fprintf(), Introduction is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

15.2: fprintf() to a file

The fopen and fclose Functions

from Serhat Beyenir, 1.4.

The fopen command is used to open or create a file. The basic syntax for fopen is as follows:
fid = fopen(filename, permission)

For example, fo = fopen('output.txt', 'w'); opens or creates a new file named output.txt and sets the permission for writing. If the file already exists, it discards the existing contents.
The fclose command is used to close a file. For example, if we type in fclose(fo);, we close the file that was created above. (It is necessary to close every file that is opened.)

.

The fprintf Function

from Serhat Beyenir, 1.4.

The fprintf function writes formatted data to the computer monitor or a file. This command can be used to save the results of a calculation to a file. To do this,

- First we create or open an output file with fopen
- Second we issue the fprintf command
- Then we close the output file with fclose.

The simplified syntax for fprintf is as follows:
fprintf=(fid, format, variable1, variable 2, ...)

✓ Example 15.2.1 Bottle Volume

Start a .m file named AcetyleneBottleInteractiveOutput.m with these lines:

```
% This script computes the volume of an acetylene bottle

% user is prompted to enter
% a radius r for a hemispherical top
% a height h for a cylindrical part
clc % Clear screen
disp('This script computes the volume of an acetylene bottle:')
disp(' ') % Display blank line

r=input('Enter the radius of acetylene bottle in meters ');
h=input('Enter the height of cylindrical part of acetylene bottle in meters ');
Vol_top=(2*pi*r^3)/3; % Calculating the volume of hemispherical top [m3]
Vol_cyl=pi*r^2*h; % Calculating the volume of cylindrical bottom [m3]
Vol_total=Vol_top+Vol_cyl % Calculating the total volume of acetylene bottle [m3]

disp(' ') % Display blank line
str = ['The volume of the acetylene bottle is ', num2str(Vol_total), ' cubic
meters.'];
disp(str);
```

Add the following lines to your .m file, in order to write to a file:

```
fo = fopen('output.txt', 'w');
fprintf(fo,'The radius of acetylene bottle: %g meters \n', r);
fprintf(fo,'The height of cylindrical part of acetylene bottle: %g meters \n', h);
fprintf(fo,'The volume of the acetylene bottle: %g cubic meters. \n', Vol_total);
fclose(fo);
```

Here, we first create the `output.txt` file that will contain the following three variables `r,` `h` and `Vol_total` . In the `fo` output file, the variables are formated with `%g` which automatically uses the shortest display format. You can also use `%i` or `%d` for integers and `%e` for scientific notation. In our script above, the `\n` (newline) moves the cursor to the next line.

Solution

Upon running the .m file, the output.txt file will display the following. The values will depend on the values you input.

```
The radius of acetylene bottle: 0.3 meters
The height of cylindrical part of acetylene bottle: 1.5 meters
The volume of the acetylene bottle: 0.480664 cubic meters.
```

.

✓ Example 15.2.2 Write a matrix to a file as a formatted table

By Carey A. Smith

```
%% Create a matrix showing the conversion of feet to inches.
feet = 0:2:10;
inches= feet*12;
matrix_feet_inches = [feet' , inches'];

% Display it on the screen, 2 methods:

%% Formatted Table with a header, method 1: for loop
fprintf('\n') % print a blank line
disp(' Feet Inches')
disp(' ----- ------')
[nrows, ncols] = size(matrix_feet_inches);
for n = 1:nrows
fprintf('%8.1f %8.2f \n', matrix_feet_inches(n,:))
end

%% Formatted Table with a header, method 2: Print the matrix with a single statement
fprintf('\n') % print a blank line
disp(' Feet Inches')
disp(' ----- ------')
fprintf('%8.1f %8.2f \n', matrix_feet_inches')
% The transpose is needed, becuz Matalb reads the data down columns
% but prints across rows.
% fprintf statment only 2 prints 2 values at a time.
% The format string is repeated until all of the data is printed.

%% Write the matrix to a file as a formatted table:
file_id1 = fopen('Formatted_Table.txt','wt')
% fopen() opens a file.
% file_id1 is the file's handle
% 'r' = read
% 'w' = write (Create the file, if it doesn't exist)
% 'a' = append to the end of the file, leaving any existing data
% 't' = text format
```

```
cnt1= fprintf(file_id1,' Feet Inches\n')
% cnt1 = the number of characters written each time.
cnt2= fprintf(file_id1,' ----- ------\n')
cnt3= fprintf(file_id1, '%8.1f %8.2f \n', feet, inches)
fclose(file_id1) % Close the file with finished writing
```

Solution

Add example text here.

.

? Exercise 15.2.1 fprintf exponential table

By Carey A. Smith

Study example in the attached file fprintf_to_file_exponential_table.m

Then modify the example in these ways:

1. In parts A & B change the format strings, so that the x column is printed 1 decimal place instead of 2 and change the exp(x) column to be pprinted with 6 decimal places instead of 8.

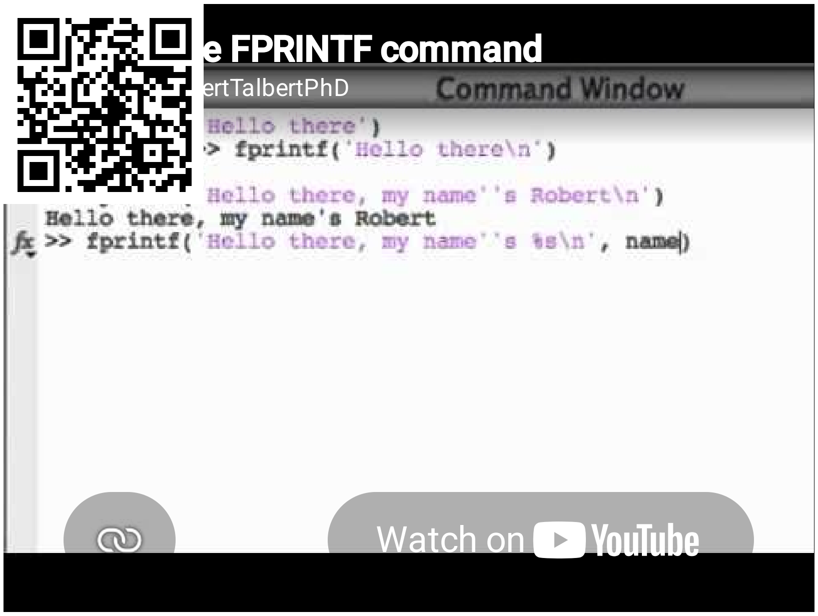
2. In part B, write to the table a file with a name like this: 'exp_table1_YourName.txt'

Answer

Add texts here. Do not delete this text first.

.

These videos may be of help:


e FPRINTF command
ertTalbertPhD
Command Window
Watch on YouTube

```
Hello there')
> fprintf('Hello there\n')

Hello there, my name''s Robert\n')
Hello there, my name's Robert
>> fprintf('Hello there, my name''s %s\n', name)
```

.

torial showing the use of fprintf us
labmarina

```
ntf('Bob Smith %d %d', 5, 5000)
th 5 5000>>
```

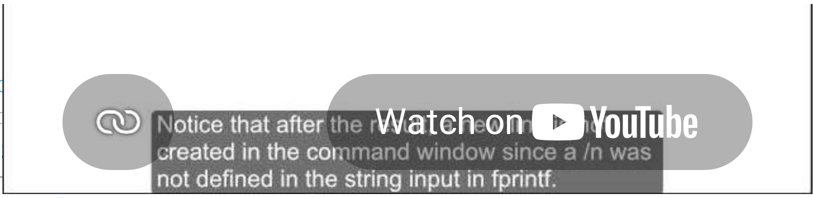
.

More information is at:

https://www.mathworks.co

https://www.mathworks.co

https://www.cs.utah.edu/~



Notice that after the [illegible] created in the command window since a /n was not defined in the string input in fprintf.

This page titled 15.2: fprintf() to a file is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith via source content that was edited to the style and standards of the LibreTexts platform.

- **1.4: Writing Scripts to Solve Problems** by Serhat Beyenir is licensed CC BY 4.0. Original source: https://cnx.org/contents/qTr2bTqz11.8:IZLgFOgb3.

15.3: Reading from an Excel file in MATLAB and Octave

By Carey A. Smith

MATLAB

MATLAB can read data from Excel files using the xlsread() command.

It helps if your are familiar with Excel's addressing of cells by columns (letters) and rows (numbers).

The xlswrite() command can be used to write to an excel file.

✓ Example 15.3.1 Read NBA Excel File

The attached Excel file, NBA_stats2016raw.xls, has statistics for the 30 NBA basketball teams.

This Matlab program reads this file and processes some of the data:

```
%% NBA_Excel_read.m
clear all; close all; clc; format compact
% Read the .xls file of NBA statistics
[nums,chars,raw] = xlsread('NBA_stats2016raw.xlsx');

% 3 matricies are returned.
% nums is a 31x26 matrix of just the numerical data. Since the first row has column headers, it is not part of the nums matrix.

% chars is a matrix of just the character data.

% raw is a matrix all of the data

% Extract the column of points for the 30 teams.
points_tot = nums(1:30, 25);
% Column 25 has the total points in the season
% The last row (row 31) has averages of the data,
% so it is not extracted.

% Compute the min, median, and max values of points_tot
points_min = min(points_tot)
points_median = median(points_tot)
points_max = max(points_tot)
```

Solution

Add example text here.

MATLAB can also write to files in the Excel format using the xlsopen() and xlswrite() functions. However, that is not discussed here.

Octave

The xlsread function is not part of the default Octave in my current version: GNU Octave Version: 7.1.0. It is planned to become part of the default Octave. Until it becomes part of the default package, you need to download and install the Octave io package from this link:

https://octave.sourceforge.io/io/

In needs to be installed into the GNU Octave program folder. It may expect to have Excel installed.

The download file is a zipped file with a .gz extension. This can be opened with various zip applications, including 7-zip, which is a free application.

https://www.7-zip.org/

This is a link has detailed instructions on how to use the Octave input/output (i/o) package:

https://wiki.octave.org/IO_package (Links to an external site.)

The basic functions--xlsopen(), xlsread(), xlswrite()--work the same in both Octave and Matlab. These are used in scripts.

Both Octave and Matlab also have interactive I/O functions.

If you have trouble using Octave's xlsread program, talk to your instructor.

This page titled 15.3: Reading from an Excel file in MATLAB and Octave is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

15.4: Chapter 15 Videos English and Espanol

15.4.1: English

TBD

15.4.2: Espanol

Impresiones - 13 - Octave

presiones - 13 - Octave
siored
13 Impresión
Watch on YouTube

Impresión Arreglos - 14 - Octave

presión Arreglos - 14 - Octave
siored
14 Impresión
Arreglos
Watch on YouTube

Escribir archivo - 15 - Octave

cribir archivo - 15 - Octave
siored
tave
15 Escribir
Archivos
Watch on YouTube

Leer Archivo - 16 - Octave

er Archivo - 16 - Octave
siored
tave
16 Leer
Archivos
Watch on YouTube

This page titled 15.4: Chapter 15 Videos English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

16: MATLAB Structures and Data Types

16.1: Numeric data types
16.2: Comparing Character Strings
16.3: table Data Type
16.4: struct Data Type
  16.4.1: struct Examples and Exercises
16.5: cell array
16.6: Chapter 16 Data Types Videos in English and Espanol

This page titled 16: MATLAB Structures and Data Types is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

16.1: Numeric data types

By Carey A. Smith

Read the MATLAB help for Floating-Point Numbers:

https://www.mathworks.com/help/matlab/matlab_prog/floating-point-numbers.html

Read the help for both double- and single-precision Floating-Point Numbers at this link.

This information includes the following statements:

- Because the default numeric type for MATLAB is double, you can create a double with a simple assignment statement:
- x = 25.783;
- Because MATLAB stores numeric data as a double by default, you need to use the single conversion function to create a single-precision number:
- x = single(25.783);

Also read "Largest and Smallest Values for Floating-Point Classes"

For most calculations, our computers have sufficient memory and speed to use double-precision floating point numbers.

Other numeric data types are at the following link. These include single and double-precision integer numbers. MATLAB also supports 1- and 2-byte integer and unsigned integer types for saving data memory. These are data types are primarily for interfacing with other application. They are not commonly used in MATLAB calculations

https://www.mathworks.com/help/matlab/numeric-types.html

.

Example 16.1.1 double vs. single floating point sum of fractions

The code shown here sums a lot of terms of a decreasing sequence using both single and double precision ✓ floating point numbers. There is some differences between these sums, because single-precision is less precise, and because when the terms get smaller than the last bit of single precision sum, the terms no longer change the running sum.

The relative error between the sums is a little more than 1%.

```
clear all;close all;clc;format compact
format long

%% Sum a lot of terms of a sequence using double precision floating-point:
a_double    = double(1:100:100e6);
a_double_len = length(a_double)
seq_double = 1./a_double;
a_double_sum = double(0);
for k = 1:a_double_len
  a_double_sum = a_double_sum + seq_double(k);
end
disp(a_double_sum)
% 1.143763955258335
```

```
%% Sum a lot of terms of a sequence using single precision floating-point:
a_single   = single(1:100:100e6);
a_single_len = length(a_single)
seq_single = single(1./a_single);
a_single_sum = single(0);
for k = 1:a_double_len
  a_single_sum = a_single_sum + seq_single(k);
end
disp(a_single_sum)
% 1.1286

a_single_sum / a_double_sum
% 0.98676

%% Note, the sum function appears to use double precision internally,
%  so that function is not used for this demonstration.
```

Solution

Add example text here.

.

Exercise 16.1.1 32-bit vs. 64-bit integer processing speed

In this assignment, you will compare the time it takes to many iterations of for loops using 32-bit integers and 64-bit integers.

Start a Matlab script with the following code.

```
clear all; close all; clc; format compact
% Store the maximum value of each type of integer
int8max = intmax('int8') % 127
int16max = intmax('int16') % 32767
int32max = intmax('int32') % 2147483647
int64max = intmax('int64') % 9223372036854775807

%% int32 loop
tic
result32 = int32(10);
for m = 1:4e4
for n = 2:2:127
ii = int32(n);
result32 = int32(result32*(ii+1));
result32 = int32(result32/ii);
end
end
int32time = toc % This reports the time to complete these loops
```

Write the time it took in a comment.

Then create a 2nd, similar loop for 16-bit integer code. Copy the 32-bit integer code and replace the following in each place it appears:

int32 by int64

result32 by result64

Write the time it took in a comment.

Answer

% On one PC, the 32-bit loop took about 0.1 seconds and the 64-bit loop took about 10 seconds.

% This shows that 32-bit integer computations are much faster than 64-bit integer computations.

.

Exercise 16.1.2 32-bit vs. 64-bit floating-point processing speed

In this assignment, you will compare the time it takes to many iterations of for loops using 32-bit floating-point floating-point and 64-bit floating-point values.

Start a Matlab script with the following code:

```
clear all; close all; clc; format compact; format long
% Store the maximum value of each type of floating-point number
float32max = realmax('single') % 3.4028e+38
float64max = realmax('double') % 1.7977e+308

%% Single loop
tic
for m = 1:10e6
result32 = single(10);
for n = 2:2:127
ii = single(n);
temp1 = single(result32*(ii+1));
temp2 = single(temp1/ii);
result32 = temp2;
end
end
result32 % echo the answer
singletime = toc
```

Then create a 2nd, similar loop for 64-bit floating-point code. Copy the 32-bit code and replace the following in each place it appears:

single by double

result32 by result64

Answer

% On one PC, the two loops took about the same amount of time, so there is not a large computational speed advantage to 32-bit floating point Matlab code.

% This implies that Matlab likely performs much of these computations in 64-bit floating-point, then converts the results to 32-bit.

% However, in C and assembly language, the single-precision code is run using single-precision operations, so it is much faster than double-precision code.

.

This page titled 16.1: Numeric data types is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

16.2: Comparing Character Strings

Comparing strings is different than comparing numbering. Comparing strings requires specific functions. In this section, tf is a logical variable meaning true or false.

Consider two character string variables. Two common string comparison functions are:

tf = strcmp(str1, str2); % case sensitive
tf = strcmpi(str1, str2); % case insensitive

These functions look for a match between strings of the same length. The user can specify to compare just the first n characters.

Study the attached file: String_compare_demo.m

❓ Exercise 16.2.1 Compare names

Create a Matlab script that does the following:
%% A. [1 pt] Create this array of names:
names = ['Alex';
'Gabe';
'Joan';
'John';
'Mary';
'Pete';
'Rosa';
'Tina']

%% B. [1 pt] Ask the user to enter a 4-letter name with this code:
name_in = input('Enter a 4-letter name: ','s');

%% C. [1 pt] Use length() to determine the length of the names array.

%% D. [5 pts] Write a for loop with loop index k to search for match of name_in
% to one of those in the names list. Use the following to find the match:

tf = strcmpi(name_in, names(k,:)); % case insensitive comparison

% When a match is found, display the index of that name in the list and break the loop
% Use disp() with num2str() to display the index with this code:
disp([name_in,' is number ',num2str(k),' in the list.'])

Extra Credit (2 pts) If a name is entered that is not in the list, figure out a way to send a message that the name is not in the list.

Are are 2 possible strategies:

Strategy A. Set a variable called valid_name = 0 before the loop.

Inside the loop, when a match is found, set valid_name = 1.

After the loop, if valid_name == 0, display a message that the name was not found.

Strategy B. Add an invalid name to the list, such as 'xxxx'. So, the length of the array will be 1 more than the number of valid names.

After the loop, if the loop index == the length of the array, than display a message that the name was not found.

Answer

The answer is not shown here.

Other string functions are:

tf = strncmp(s1,s2,n): Compare first n characters of strings (case sensitive)

tf = strncmpi(s1,s2,n): Compare first n characters of strings (case insensitive)

tf = strfind(str,pat): Find strings within other strings

.

This page titled 16.2: Comparing Character Strings is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

16.3: table Data Type

By Adam L. Lambert, "Zero to Matlab", section 12.1

The "table" data type is only available in MATLAB; it is not implemented in Octave.

Of the complex data structures, tables are probably the easiest to learn. You can think of a table as a spreadsheet. It looks a lot like a 2-D array, but the columns are labeled with text. Tables are the default structure used when you import data (such as a .CSV file) through the GUI.

Let's build a simple table to see how they work.

When creating a table in MATLAB, the arrays must be in column format. With that in mind, let's make up a sample data set. I'm just going to do this in the Command Window, but feel free to use a script.

```
>> Temp=[100; 200; 300; 400; 500]
```

Note the semicolons in the array, this vertical concatenation creates a column vector.

```
>> Rate = [2.1; 2.5; 2.8; 3.4; 3.6]
```

Now we can use the table function to create a table and assign it to a variable name.

```
>> T = table(Temp,Rate)
```

Without a semicolon after the table command, the table should be displayed in the Command Window. It shows a header in addition to the data. You should also see a new variable, T, in the Workspace of class table. It should have five rows and two columns.

We can use dot syntax to access individual columns in the table.

```
>> T.Temp
```

This array can also be indexed to get a single value.

```
>> T.Temp(2)
```

At this point, we can easily plot the columns of the table using readable syntax.

```
 figure;


>> plot(T.Temp,T.Rate,'o')
```

Values can be added to the table just like an array. However, MATLAB will automatically extend the rows of the other columns and fill them with default values. It will also issue a warning. Try this command.

```
>> T.Temp(6) = 600
```

Now let's overwrite the rate.

```
>> T.Rate(6) = 4.2
```

There is one issue that must be pointed out. All of the variable types in a column must match. In other words, you cannot mix doubles and logicals in a single column. If we try to add a char to the end of one of our columns, then we will get an error. Try it out.

```
>> T.Temp(7) = 'Hot!'
```

That error is a little cryptic huh? In this case, MATLAB sees an array of chars while every other entry is a single value. So what happens if we try to assign it a single letter?

```
>> T.Temp(7) = 'H'
```

Now we have a number (72), but not a number that is relevant to our data. This is the ASCII code that MATLAB uses for the character H. Since MATLAB had already decided to store numbers in that column, it automatically converted the character to a number. You can do this yourself using the uint8 function.

```
>> uint8('H')
```

If we want to store strings in a table, then we need a whole column of strings. The best way to get that is using a cell array, which will be discussed in another section.

As mentioned at the beginning of the chapter, if you use the GUI to import a data set such as a .CSV file, a table will be the default suggestion. You can choose other variable types, but often a table is the easiest method to script around. Generally speaking, as long as your data is clean, then MATLAB will select the correct variable type for each column.

This page titled 16.3: table Data Type is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

16.4: struct Data Type

By Adam L. Lambert, "Zero to Matlab", section 12.

Structs are more complex than tables. In this section we'll walk through some basic examples to give you a feel for how they behave. Again I will work in the Command Window. Structs store values in elds. Each field has a name. We access the fields using the same dot notation that was used in the table section.

The simplest method for creating a struct is to define it using dot notation.

```
>> c.a = 1
```

Now we have a struct named c in the workspace. We can see in the Command Window that it has a field named "a" which has a value of 1. Now add another field:

```
>> c.b = 2
```

Now the struct has two fields, a and b.

We can use the value from a field in an expression like this.

```
>>  y = 2*c.b
```

The fields of a struct can be any variable type. Let's add an array to c.

```
>> c.temp = [100,200,300]
```

And now a string.

```
>> c.name = 'Harvey'
```

This is the complete structure:

```
 c.a = 1
 c.b = 2
 c.temp = [100,200,300]
 c.name = 'Harvey'
```

We can actually store an array of structs. Let's add a second element.

```
>> c(2).a = 3
```

Now in the Command Window you will see that we have a 1x2 struct array and the name of each field is reported. Since we have only assigned on field in the second element, then the other fields will be blank.

Check for yourself.

```
 c(2).name
```

Display the completed structure, c:

At this point we need to discuss two issues that can trip people up. The first is that the fields in each element do not have to be the same variable type. I can assign a string to c.temp and a double to c.name.

```
>> c(2).temp = 'Hot!'
```

Now print the temp field for the whole array in the Command Window.

```
>> c.temp
```

First we see the temperature array assigned to the default variable name. Next we see the string. This is because MATLAB is looping through the struct and printing them one at a time. If we try to assign it to a variable then we will only get the first element. In this case it will be the array.

```
>> y = c.Temp
```

If we want the field from the second element, then we have to index the struct.

```
>> y = c(2).temp
```

The other issue with struct arrays is that we cannot use them directly in calculations. For example, the a field has only individual numbers in both elements. However the following command will generate an error.

```
>> y = 2*c.a
```

This error tells us that we are using too many numbers with the multiplication operator. Again, the struct must be indexed for this calculation to work.

```
>> y = 2*c(1).a
```

This page titled 16.4: struct Data Type is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

16.4.1: struct Examples and Exercises

By Carey A. Smith

✓ Example 16.4.1.1 Structure for simulation of the effect of an earthquake

```
% The magnitude 6.7 Northridge, California, earthquake of 17 January 1994
% https://pubs.er.usgs.gov/publication/70017610
Earthquake.name = 'Northridge';
Earthquake.year = 1994;
Earthquake.mag = 6.7;
Earthquake.length = 15; % km
Earthquake.vertical = 70; % cm max vertical movement
% https://en.Wikipedia.org/wiki/1994_N...dge_earthquake
Earthquake.duration = 20; % seconds
Earthquake.accel_g =1.82; % g
Earthquake.accel_ms2=Earthquake.accel_g*9.81; % m/s^2
% Simulation parameter
Earthquake.scale = 1.4; % Scale factor for simulations
```

Solution

Add example text here.

.

? Exercise 16.4.1.1 Basketball Team

1. Create a struct for a team of 5 basketball players. They may be men or women, real or fictional. For each member of the team create struct called "player".
2. Create the following fields in the "player" structure for each player.:
    1. Name (a string)
    2. Position (Choose one of these ‘Center’, ‘Power Forward’, ‘Small Forward’, ‘Shooting Guard’, ‘Point Guard’)
    3. Height in inches (not feet) Example: 75
    4. Shirt number
3. Fill in you teams structure array with a separate sub-structure for each of the 5 starting players.
4. Example. Suppose you had already filled out the structures for the 1st 4 players. Then the 5th substructure could be defined as follows:

```
player(5).Name = 'Kareem';
player(5).Position= 'Center';
player(5).Height = 84;
player(5).Shirt = 33;
```

5. Create a header with this line:

```
fprintf(' Name Position Height Shirt\n') % \n creates a "new line"
```

6. Create a “for loop” that goes from 1 to 5 and that displays each player’s information on one line. Each field should be separated by a tab “\t” or by 4 spaces. Use \n at the end of the format string to put a new line after each player's info.

Example for loop. (The tabs and/or spaces could be changed, but the for loop must go from n=1 to n=5)

```
for n=1:5
    fprintf('%13s \t%14s \t%i \t %i\n', ... % \t inserts a tab
    player(n).Name, player(n).Position, ...
    player(n).Height, player(n).Shirt)
end
```

Answer

Create your own struct.

.

✓ Example Extra close brace or missing open brace Triangle struct and draw

This example creates a triangle struct and uses it to draw 2 right triangles on one figure.

The first routine sets the parameters of each triangle.

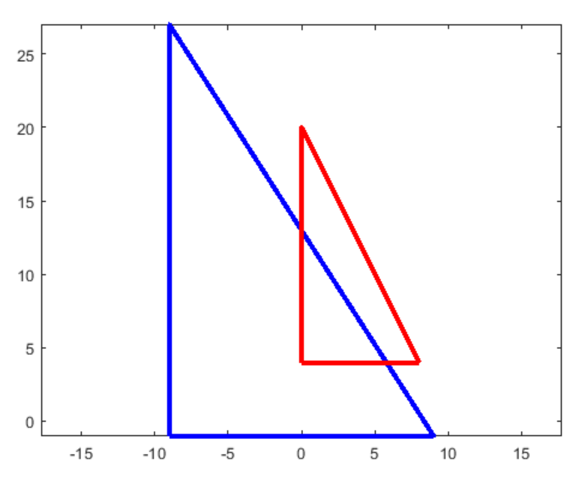
The second file is a function that draws a triangle using the parameters in the struct. Only one argument, the struct, is passed to the function. Thus, a struct can group related variables and simplify function calls.

```
%% Triangle structure demo
clear all, format compact, format shortg; close all; fclose all; clc;
% 1st triangle
Tri1.pos = [-9,-1];
Tri1.width = 18;
Tri1.height= 28;
Tri1.color = 'b';
%% Double-click on a structure in the Workspace to see the details

figure;
draw_triangle(Tri1);
axis equal

%% 2nd triangle
Tri2.pos = [0,4];
Tri2.width = 8;
Tri2.height= 16;
Tri2.color = 'r';

hold on;
draw_triangle(Tri2);

function [] = draw_triangle(Tri)
% Draw a right triangle
x = zeros(1,4); % Need 4 points (1st & last are the same)
y = x;
% 1st point
x(1) = Tri.pos(1);
y(1) = Tri.pos(2);
% 2nd point: to the right, with the same y as point 1
x(2) = x(1) + Tri.width;
y(2) = y(1);
% 3rd point: up, with the same x as point 1
x(3) = x(1);
y(3) = y(1) + Tri.height;
% Last point = 1st point
x(4) = x(1);
```

```
y(4) = y(1);
% Debug
x = x
y = y
plot(x,y,Tri.color,'Linewidth',3);
```

Solution

This is the resulting plot:



.

Exercise 16.4.1.2 Rectangle struct and draw

The logic and code for this exercise is similar to the

1. [5 pts] Create 3 rectangle structures that specify the position, size, and color of 3 rectangles with these characteristics:

```
% Rectangle 1
rect1.pos= [30,20]; % x and y coordinates of the lower left corner
rect1.width = 500;
rect1.height= 400;
rect1.color = 'b';
% Rectangle 2
rect2.pos= [350,300]; % x and y coordinates of the lower left corner
rect2.width = 220;
rect2.height= 250;
rect2.color = 'r';
% Rectangle 3
rect3.pos= [300,250]; % x and y coordinates of the lower left corner
rect3.width = 150;
rect3.height= 400;
rect3.color = 'c';
```

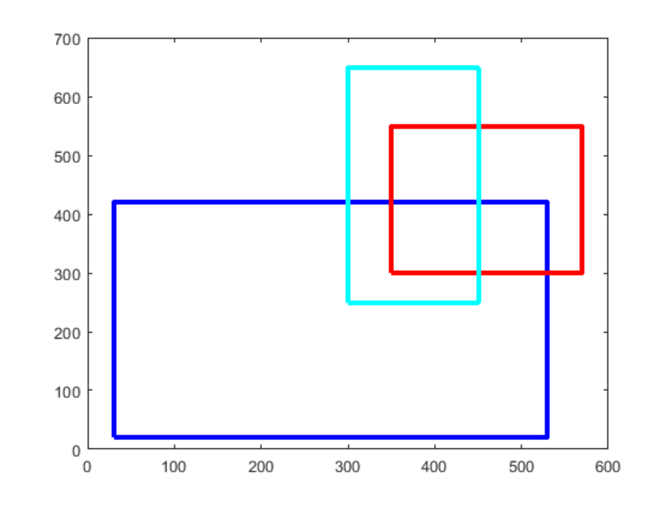
2. [7 pts] Write a draw_rectangle(rect) function whose uses a rect structure to plot 1 rectangle. Since variables in a function are local, you can use "rect" for the structure name inside the function. You don't need to use rect1, rect2, rect3.

3. [3 pts] Open a figure, use ”hold on;” and plot these 3 rectangles by calling your rectangle plotting function with each of the 3 rectangle structures.

- Helps:
- Like the draw_triangle function, the draw rectangle function needs to create an x vector & a y vector with the coordinates of the corners. In order to get back to the starting point & close the rectangle, you will need to have 5 pairs of (x,y) coordinates, with the last coordinate being the same as the first coordinate.
- The 1st corner is like home plate. It is in the lower-left corner.
- The 2nd corner is like 1st base. It’s x-coordinate is the 1st corner’s x-coordinate + the width of the rectangle. It’s y-coordinate is the same as the 1st corner’s .
- The 3rd corner is like 2nd base. It’s x-coordinate is the same as the 2nd corner. It’s y-coordinate is 2nd corner’s y-coordinate + the height of the rectangle.Figure out the coordinates of 3rd base. The last line goes from 3rd base to home plate.
- To plot each of the 4 lines, the 1st argument of plot() is a vector of the X coordinates.The 2nd argument of plot() is a vector of the Y coordinates.

Answer

The resulting plot should look like this:



Add texts here. Do not delete this text first.

.

This page titled 16.4.1: struct Examples and Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

16.5: cell array

By Carey A. Smith

An ordinary array of strings can be created when all of the strings are the same length.

Cell arrays are useful for containing multiple character strings of different lengths.

✓ Example 16.5.1 ✓ String Array Shape Comparison

```
%% String_array_comparison_demo
clear all, format compact, format shortg; close all; fclose all; clc;
% A string array can only be created in MATLAB if every string is the same length.
% If you try to create an array with strings of different lengths, MATLAB generates an error.
% Octave, however, will automatically pad the short strings with blanks, in order to avoid an error.
% The shapes in this example are listed in order of the number of sides.
% All the strings have been abbreviated to 3 characters,
% so that an array can be created.
shapes = ['cir' % circle (1 side)
'ban' % banana (2 sides)
'tri' % triangle (3 sides)
'squ' % square (4 sides)
'pen' % pentagon (5 sides)
'hex']; % hexagon (6 sides)
[n, m] = size(shapes)
%% Input shape name.
disp('Shapes to choose from:')
disp('triangle')
disp('square')
disp('hexagon')
disp('circle')
disp('banana')
in_shape = input('Enter the name of a shape (without quotes): ','s')
%% Match the input to the shapes array
tf = 0; % True/False
for k = 1:n
% Compare the first 3 letters of the input shape name
% to each of the names in the shapes array.
tf = strcmpi(in_shape(1:3), shapes(k,:)); % case insensitive
if(tf == 1) % If True
disp(['A ',in_shape,' has ',num2str(k),' sides.'])
break;
end
end
```

```
if(tf == 0)
disp(['No match was found for ',in_shape])
end
```

Solution

Add example text here.

.

Cell arrays can contain character strings of different lengths.

Cell arrays are defined by using { } ("squiggly brackets", a.k.a. "curly brackets". a.k.a. "curly braces")

The following example demonstrates how to create and use a cell array.

✓ Example 16.5.2 Cell Array Name Comparison

```
%% Cell_array_demonstration.m
clear all, format compact, format shortg; close all; fclose all; clc;
% MATLAB "cell arrays" allow character strings of various lengths
% to be stored in the same array.

%% This is an example of how to create a cell array.
% Note that cell arrays use squiggly brackets { }, instead of
% the square brackets [ ], used by numerical arrays.
names = {'Steven'
'Matt'
'Giovanni'
'Ashanti'
'Meagan'
'Gabriela'
'Peggy'}
names_len = length(names);
%% This shows how a cell array can be used
% [1 pt] Ask the use to enter a name
name_in = input('Enter a name from the list above (without quotes): ','s');

%% Search for match of name_in to one of those in the names list.
tf = 0; % True/False
for k = 1:names_len
name_k = names{k};
tf = strcmpi(name_in, name_k); % case insensitive comparison
if(tf == 1) % If True
disp([name_in,' is number ',num2str(k),' in the list.'])
break;
end
end
if(tf == 0)
disp(['No match was found for ',name_in])
end
```

Solution

Add example text here.

Cell arrays can also be used as parts of structures.

Exercise 16.5.1 Cell Array of South American Countries

Make a cell array called South_American_Countries.

You can find a link to these countries at List of South American Countries (Links to an external site.)

In this list, put the countries called "Sovereign states" in your cell array. Do not include the territories, which are listed below the Sovereign states.

Then add this code to display the 3rd and 7th countries in your cell array:

South_American_Countries{3} % Note that you need squiggly brackets to access members of a cell array.
South_American_Countries{7}

(If your cell array has a different name, use that name.)

Answer

The answer is not given here.

.

Cell arrays can be used to create a two-lin title, as shown in this example:

Example 16.5.1 Cell Arrays in Figure Titles

```
%% Two-ltne title with a cell array in MATLAB. (Doesn't work in Octave.)
% Mimic 70 dice rolls.
% Create a vector with uniform random numbers between 1 and 6.
nn = 70
dd = randi(6,1,70); % dice roll values

% Plot a histogram
figure;
bin_edges= (1:7)-0.5;
histogram(dd,bin_edges);
%Compute the mean of the vector.
mdice = mean(dd)
%Compute the standard deviation of the vector.
sdice = std(dd)

% Create a two line title using 2 cell arrays
title({[num2str(nn),' Dice Rolls']}, ...
{['Mean= ',num2str(mdice),', Std. Dev.= ',num2str(sdice)]});
```

Solution

Add example text here.

.

This page titled 16.5: cell array is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

16.6: Chapter 16 Data Types Videos in English and Espanol

English

atlab: Structs
l Holtschulte
structs
MATLAB
Watch on YouTube

.

atlab: Cell Arrays
l Holtschulte
ll Arrays
MATLAB
Watch on YouTube

..

.

Espanol

ATLAB PARA ALGEBRA LINEAL 01
as
IA
TLAB
CON ENFASIS
EN ALGEBRA LINEAL
Watch on YouTube

.

This page titled 16.6: Chapter 16 Data Types Videos in English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

17: Top-Down Design, Agile Software Development, and Object-Oriented Programming in MATLAB

17.1: Top-Down Design
17.2: Agile Software Development
17.3: Object-Oriented Programming, Part 1
17.4: Object-Oriented Programming, Part 2
17.5: Chapter 17 Videos in English and Espanol

This page titled 17: Top-Down Design, Agile Software Development, and Object-Oriented Programming in MATLAB is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

SECTION OVERVIEW

17.1: Top-Down Design

Creating a medium to large software project requires good planning, especially when working in a team. The standard approach is called "Top-Down Design". This can also be applied to projects using a combinations of hardware and software. This is illustrated by this figure from Wikipedia:

(https://en.wikipedia.org/wiki/V-model_(software_development)

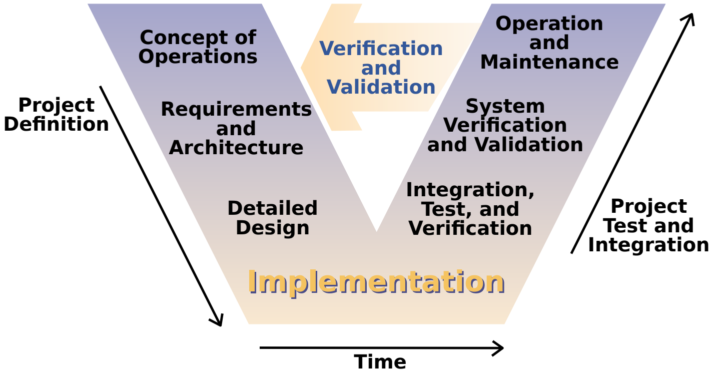

Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

Figure \(\PageIndex{1}\): The V-Model of the Systems Engineering Process

(This work is in the **public domain** in the United States because it is a work prepared by an officer or employee of the United States Government as part of that person's official duties under the terms of *Title 17, Chapter 1, Section 105 of the US Code*.

https://commons.wikimedia.org/wiki/F...Process_II.svg)

The V-model of engineering and software development approach has the following steps:

- Concept of Operation. This is the basic concept of what the final software or device should do.
- Requirements and Architecture (a.k.a. Requirements Analysis) This refines the concept of operation to specific requirements, to what could be contract requirements.
  - Example 1: A navigation system for a commercial airline. The requirements would specify the power available, the inputs from other devices (e.g. altimeter), the outputs (roll, pitch heading, speed, rotation rates, latitude, longitude, altitude), the rates of the inputs and outputs, and the connections. I would not specify the insides of the navigation box.
  - Example 2: Process for a census. The requirements would specify the data desired from each household, such as names, ages, address, employment or retirement or looking for work, or disability status etc. It would specify what the user interface would look like for processing the data after it has been collected.
  - Example 3: Software for an ATM machine. This would specify the user interface and required functions, such has handling withdrawals and deposits, allowing for checking and savings accounts, ability to check balances, and an option for a receipt.
- System design.
  - Example 1: Navigation system. The System Design would specify what type of gyros, accelerometers, and other sensors to be used, the number of circuit boards, the microprocessors, and the high-level algorithms.
  - Example 2: Census. The System Design would specify whether the data will be collected electronically and/or by paper. It would specify how the data will be stored (database or spreadsheet software), what the categories are, what the data summaries should look like. It could specify the need to check for duplicate entries or for missing data from known apartments or zip codes.

    - Example 3: ATM Software. This would specify the software interfaces with the card reader and the cash dispenser and the check scanner. It would include access to the user's account and the related ID verification and security software. It could include interfaces with partner banks or credit unions.
- Software detailed design. This specifies the software algorithms, precision needed, interfaces with the hardware, etc.
  - Example 1: Navigation system. The Detailed design would specify the software functions needed and the algorithms for them. If there are multiple processors, it would specify which functions are implemented on which processor. It would include estimates of the time for each function and and timeline so that the outputs can be computed at the required rates.
  - Example 2: Census. This would specify how the data will be processed and what subroutines are needed. Routines might be designed for common interfaces with different data sets. It might include how to merge electronic and paper data.
  - Example 3: Software for an ATM machine. This would specify the algorithms for each of the functions.

Once the software is written it needs to be tested. This is called verification. First, each function is checked for correct operations with both valid and invalid inputs. When errors are found, then they are corrected or the design and/or requirements are reviewed for problems and possibly updated.

Also, functions with hardware interfaces are tested with their hardware.

Then groups of relation functions are tested together to verify that their interfaces work as designed and the results are correct. Again, corrections are typically needed. These may be corrections in the software or the design or sometimes the requirements need to be adjusted/clarified.

See also:

V-Model of Engineering Development https://en.Wikipedia.org/wiki/V-Model_(software_development)

.

This page titled 17.1: Top-Down Design is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

17.2: Agile Software Development

By Carey Smith

Agile Software Development

The Top-Down Design Method works well when the requirements are well-defined.

Sometimes the general goal of the project or the customer is understood, but the details are not well defined. Perhaps there are multiples ideas and you want to explore some of each. Or perhaps a new algorithm is being considered or a new piece of hardware is needed.

I these case, agile software development can be an advantage. The concept is to start working on one or more pieces and have frequent contact with the customer or management to see if the those parts are satisfactory or if changes need to be made or alternatives explored. This may lead to a better product and/or more satisfied customers and management.

This page titled 17.2: Agile Software Development is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

17.3: Object-Oriented Programming, Part 1

Object-Oriented Programming is an important concept in modern programming.

It is what distinguishes C++ from the C programming language.

The fundamentals concepts are:

* Definition of data classes (types)

* Storage of data classes

* Methods (functions) that implement operation on each data class.

MATLAB uses this approach hidden from the user. When variables, vectors and arrays are created, Matlab keeps track of the data type and size/dimensions. Number types include character string or number, floating-point or integer, single or double precision, real or complex. Most of its routines can handle scalars, vectors, and arrays, without the typical user needing to know the details that are under the covers.

MATLAB also allows the user to define his/her own objects with the object's data type and routines. An example is a "student" data type that has character string data, such as name, major, and contact info. It also has numerical data such as number of credits passed in each general education area and in the dstudent's major.

A short tutorial on Object-Oriented Programming in MATALB is given in this on-line course:

MATLAB: An Introductory Course (Links to an external site.)

You can get to the Object-Oriented tutorial by clicking on the "Table of Contents" tab, then selecting "Object-Oriented Programming (OOP)". Or you can search of "Object-Oriented Programming". It's about 85% of the way down the page.

Here are more links for Matlab's Object-Oriented capabilities:

Introduction to Object-Oriented Programming in MATLAB (Links to an external site.)

Object-Oriented Programming in Matlab examples

This page titled 17.3: Object-Oriented Programming, Part 1 is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

17.4: Object-Oriented Programming, Part 2

Author: Randy Matusky

This is from section "Object-Oriented Programming (OOP)" of this link:

https://oercommons.org/authoring/4896-matlab-an-introductory-course/view

17.4.1: Introduction to OOP

Object-oriented programming (OOP) applies to the development of software or “objects” that have data fields (attributes that describe the object) and associated procedures known as methods. Objects are composed of two elements: data, such as numbers, strings or variables and actions such as functions. You can then combine objects or have them interact together in order to design applications and computer programs. Examples of object-oriented programming languages include Objective-C, Smalltalk, Java and C#.

17.4.2: Classes

One of the first distinctions you must make before trying to create programs in MATLAB is the difference between classes and objects. A class is a definition that specifies certain characteristics that all instances of the class share. These characteristics are determined by the properties, methods and events that define the class and the values of attributes that modify the behaviour of each of these class components. Class definitions describe how objects of the class are created and destroyed, what data the objects contain, and how you can manipulate this data.

Objects contain actual data for a particular entity that is represented by the class. An example to help better explain the difference is to think that a *class* is like a bank account, while an *object* is one specific bank account that includes a real account number.

The following table will help you better understand other common definitions used in MATLAB.

Term	**Definition**
Class Definition	Description of what is common to every instance of a class.
Properties	Data storage for class instances.
Methods	Special functions that implement operations that are usually performed only on instances of the class.
Events	Messages that are defined by classes and broadcasts by class instances when some specific action occurs.
Attributes	Values that modify the behaviour of properties, methods, events, and classes.
Listeners	Objects that respond to a specific event by executing a callback function when the event notice is broadcast.
Objects	Instances of classes, which contain actual data values stored in the objects' properties.
Subclasses	Classes that are derived from other classes and that inherit the methods, properties, and events from those classes (subclasses facilitate the reuse of code defined in the superclass from which they are derived).
Superclasses	Classes that are used as a basis for the creation of more specifically defined classes (i.e., subclasses).
Packages	Folders that define a scope for class and function naming.

17.4.3: **Defining A Class**

The first step to defining a class is by creating a new *m-file* with the same name as the class you want to create. You need to start the file with the **classed** keyword followed by the class name. The following line is for defining the *properties* and *methods* of the class. A special keyword called the **constructor** is needed to create or construct a new object, and it must have the same name as the class. the constructor can take any argument which specifies initial values for the properties, and must return one argument, the constructed object. Properties can optionally be assigned default values. If no default is specified and no value is assigned by the constructor, they are assigned the empty matrix.

The following is an example of how to create a class in MATLAB:

Example 17.4.1 how to create a class in MATLAB

Classes.jpg

Line	Explanation
classdef mydate	This is where you write a description of the class
function obj = mydate (minute, hour, day, month, year)	This is the class constructor

17.4.4: Solution

Add example text here.

.

17.4.5: Create Class Objects

Now that you have created the class, you can create date objects. The following two lines are examples of how you can create date objects.

d1 = mydate (0, 5, 20, 5, 1990);

d2 = mydate ();

The properties that you have just defined are *public*, meaning that they are accessible from outside the class. You can access and assign them using dot notion:

day = d1.day;

d1.year = 1990;

.

This page titled 17.4: Object-Oriented Programming, Part 2 is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by Carey Smith.

17.5: Chapter 17 Videos in English and Espanol

This page titled 17.5: Chapter 17 Videos in English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

18: Integration and Differentiation

18.1: Integration- Computing the Area Under a Curve
  18.1.1: Problem Set
  18.1.2: The integral() function
18.2: Numerical Differentiation
18.3: Chapter 18 Differentiation and Integration Videos in English and Espanol

---

This page titled 18: Integration and Differentiation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

18.1: Integration- Computing the Area Under a Curve

This chapter essentially deals with the problem of computing the area under a curve. First, we will employ a basic approach and form trapezoids under a curve. From these trapezoids, we can calculate the total area under a given curve. This method can be tedious and is prone to errors, so in the second half of the chapter, we will utilize a built-in MATLAB function to carry out numerical integration. (This was reformatted by Carey Smith. Also, the text was modified to show integrating a function that is defined in a separate file in addition to showing the use of an anonymous function.)

18.1.1: A Basic Approach

There are various methods to calculating the area under a curve, for example, Rectangle Method, Trapezoidal Rule and Simpson's Rule. The following procedure is a simplified method.

Consider the curve below:

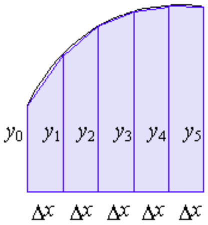


Figure 18.1.1. Numerical integration

Each segment under the curve can be calculated as follows:

$$\frac{1}{2}(y_0+y_1)\Delta x+\frac{1}{2}(y_1+y_2)\Delta x+\frac{1}{2}(y_2+y_3)\Delta x$$

Therefore, if we take the sum of the area of each trapezoid, given the limits, we calculate the total area under a curve. Consider the following example.

Example 1

Given the following data, plot an x-y graph and determine the area under a curve between x=3 and x=30

Index	**x [m]**	**y [N]**
1	3	27.00
2	10	14.50
3	15	9.40
4	20	6.70
5	25	5.30
6	30	4.50

Data Set

First, let us enter the data set.

```
x=[3,10,15,20,25,30];
y=[27,14.5,9.4,6.7,5.3,4.5];
```

When you type in `[x',y']` , you will see the following tabulated result. Here we transpose row vectors with ' and displaying them as columns:

```
  3.0000 27.0000
 10.0000 14.5000
 15.0000  9.4000
 20.0000  6.7000
 25.0000  5.3000
 30.0000  4.5000
```

To plot the data type the following:

```
figure
plot(x,y)
title('Distance-Force Graph')
xlabel('Distance[m]')
ylabel('Force[N]')
grid
```

The following figure is generated:

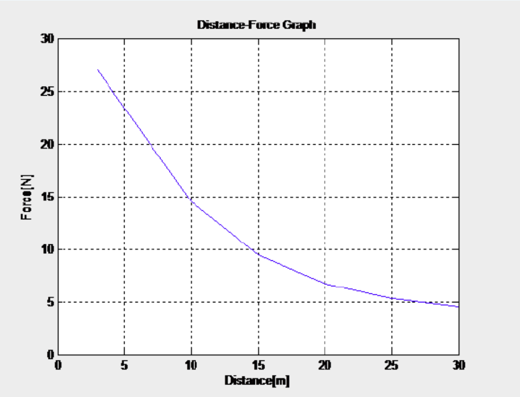


Figure 18.1.2. Distance-Force Graph

To compute dx for consecutive x values, we will use the index for each x value, see the given data in the question.:

```
dx=[x(2)-x(1)
    x(3)-x(2)
    x(4)-x(3)
    x(5)-x(4)
    x(6)-x(5)];
```

dy is computed by the following command:

```
dy=[0.5*(y(2)+y(1))
    0.5*(y(3)+y(2))
    0.5*(y(4)+y(3))
    0.5*(y(5)+y(4))
    0.5*(y(6)+y(5))];


[dx, dy] % Disply these as a table with 2 columns
```

```
7.0000 20.7500
5.0000 11.9500
5.0000  8.0500
5.0000  6.0000
5.0000  4.9000
```

Our results so far are shown below

x [m]	y [N]	dx [m]	dy [N]
3	27.00		
10	14.50	7.00	20.75
15	9.40	5.00	11.95
20	6.70	5.00	8.05
25	5.30	5.00	6.00
30	4.50	5.00	4.90

x, y and corresponding differential elements

If we multiply dx by dy, we find da for each element under the curve. The differential area da=dx*dy, can be computed using the 'term by term multiplication' technique in MATLAB as follows:

```
da=dx.*dy
da =
145.2500

 59.7500

 40.2500

 30.0000

 24.5000
```

Each value above represents an element under the curve or the area of trapezoid.

By taking the sum of array elements, we find the total area under the curve.

```
sum_da = sum(da) % 299.75
```

The following illustrates all the steps and results of our MATLAB computation.

x [m]	y [N]	dx [m]	dy [N]	dA [Nm]
3	27.00			
10	14.50	7.00	20.75	145.25
15	9.40	5.00	11.95	59.75
20	6.70	5.00	8.05	40.25
25	5.30	5.00	6.00	30.00
30	4.50	5.00	4.90	24.50
				299.75

Computation of the approximate area under a curve

18.1.2: The Trapezoidal Rule

Sometimes it is rather convenient to use a numerical approach to solve a definite integral. The trapezoid rule allows us to approximate a definite integral using trapezoids.

18.1.2.1: The trapz() Function

Z = trapz(Y) computes an approximation of the integral of Y using the trapezoidal method.

Now, let us see a typical problem.

Example 2

Given Area $= \int_2^5 x^2 dx$, an analytical solution would produce 39.

Use trapz command and solve it

Initialize variable x as a row vector, from 2 with increments of 0.1 to 5:

```
x=2:.1:5;
```

Declare variable y as `y=x^2;` .

Note the following error prompt:

```
??? Error using ==> mpower Inputs must be a scalar and a square matrix.
```

This is because x is a vector quantity and MATLAB is expecting a scalar input for y.

Because of that, we need to compute y as a vector and to do that we will use the dot operator as follows: `y=x.^2;` .

This tells MATLAB to create vector y by taking each x value and raising its power to 2.

Now we can issue the following command to calculate the first area, the output will be as follows:

```
area1=trapz(x,y)
area1 =
39.0050
```

Notice that this numerical value is slightly off. So let us increase the number of increments and calculate the area again:

```
x=2:.01:5;
y=x.^2;
area2=trapz(x,y)
area2 =
39.0001
```

Yet another increase in the number of increments:

```
x=2:.001:5;
y=x.^2;
area3=trapz(x,y)
area3 =
39.0000
```

Example 3

Determine the value of the following integral:

$\int_0^\pi \sin(x) \mathrm{d}x$

Initialize variable x as a row vector, from 0 with increments of pi/100 to pi:

```
x=0:pi/100:pi;
```

Declare variable y as

```
y=sin(x);
```

Issue the following command to calculate the first area, the output will be as follows:

```
area1=trapz(x,y)
area1 =
1.9998
```

Let us increase the increments, as above:

```
x=0:pi/1000:pi;
y=sin(x);
area2=trapz(x,y)
area2 =
2.0000
```

Example 4

A gas expands according to the law, $PV^{1.4}$=c.

Initially, the pressure is 100 kPa when the volume is 1 m^3.

Write a script to compute the work done by the gas in expanding to three times its original volume[1].

Recall that PV diagrams can be used to estimate the net work performed by a thermodynamic cycle, see Wikipedia or we can use definite integral to compute the work done (WD) as follows:

$$\mathrm{WD} = \int p \mathrm{d}v$$

If we rearrange the expression pressure as a function of volume, we get:

$$P = \frac{c}{V^{1.4}}$$

By considering the initial state, we can determine the value of c:

$$\begin{aligned} c &= 100 \times 1^{1.4} \\ &= 100 \end{aligned}$$

From the equation and the equation above, we can write:

$$P = \frac{100}{V^{1.4}}$$

By inserting P in WD, we get:

$$\mathrm{WD} = \int_1^3 \frac{100}{v^{1.4}} \mathrm{d}v$$

For MATLAB solution, we will consider P as a function of V and WD. Now, let us apply the three-step approach we have used earlier:

1. Initialize variable volume as a row vector, from 1 with increments of 0.001 to 3:

```
v=1:0.001:3;
```

2. Declare variable pressure as

```
p=100./v.^1.4;
```

3. Use the `trapz` function to calculate the work done, the output will be as follows:

```
WorkDone=trapz(v,p)
WorkDone =
88.9015
```

These steps can be combined in an m-file as follows:

```
clc
disp('A gas expands according to the law, pv^1.4=C')
disp('Initial pressure is 100 kPa when the volume is 1 m3')
disp('Compute the work done by the gas in expanding')
disp('To three times its original volume')
disp(' ') % Display blank line
v=1:.001:3; % Creating a row vector for volume, v
p=100./(v.^1.4); % Computing pressure for volume
WorkDone=trapz(v,p) % Integrating p*dv over 1 to 3 cubic meters
```

Example 5

A body moves from rest under the action of a direct force given by $F=\frac{15}{x+3}$

where x is the distance in meters from the starting point. Write a script to compute the total work done in moving a distance 10 m.[2]

Recall that the general definition of mechanical work is given by the following integral, see Wikipedia:

$$\mathrm{WD}=\int F \mathrm{d}x$$

Therefore we can write:

$$\mathrm{WD}=\int_0^{10} \frac{15}{x+3} \mathrm{d}x$$

Applying the steps we followed in the previous examples, we write:

```
clc
disp('A body moves from rest under the action of a direct force given')
disp('by F=15/(x+3) where x is the distance in meters')
disp('From the starting point.')
disp('Compute the total work done in moving a distance 10 m.')
disp(' ') % Display blank line
x=0:.001:10; % Creating a row vector for distance, x
F=15./(x+3); % Computing Force for x
WorkDone=trapz(x,F) % Integrating F*dx over 0 to 10 meters.
```

The output of the above code is:
A body moves from rest under the action of a direct force given
by F=15/(x+3) where x is the distance in meters from the starting point.
Compute the total work done in moving a distance 10 m.

```
WorkDone =
21.9951
```

18.1.3: The integral() Function

As we have seen earlier, `trapz` gives a good approximation for definite integrals. The `integral` function streamlines numerical integration even further.

(Note by Carey Smith: the original version introduces anonymous functions. It is not necessary to use anonymous functions. It is simpler to create a separate function file like was done for the fzero() function in Chapter 10. Another alternative for MATLAB, but not for Octave, is to create a subfunction at the bottom of the script.)

Syntax for `integral()`

To evaluate an integral from a minimum to a maximum value, we specify a function and its minimum and maximum

`Z = integral(@fun, xmin, xmax)` .

@fun is a function handle to the function that you want to integrate.

xmin is the lower bound of the integral and xmax is th eupper bound of the integral.

Example 6

Given `y=x^2` , evaluate the integral from x=2 to x=5 as we have done it with `trapz` command.

Define a function called "myfunction.m"

```
function y = myfunction(x)
  y = x.^2;
end
```

Test integrating this function from x = 2 to 5 with this code:

```
Z = integral(@myfunction, 2, 5)
```

You should get a result of `Z = 39` .

Notice that, unlike in the examples, we did not need to define a vector and change the increments to get an accurate result.

The integral() function determines appropriate point spacings. Typically, the spacing will not be uniform, but are optimized for speed an accuracy.

.

18.1.3.1: Anonymous Functions (Optional)

An anonymous function is a function that can be defined in the command window (i.e. it does not need to be stored in a program file). Anonymous functions can accept inputs and return outputs, just as standard functions do such as `sqrt(X)` or `log(X)` .

To define an anonymous function, first we create a handle with `@(x)` and type in the function:

`myfunction=@(x) x^2+1` .

If you want to evaluate `myfunction` at 1, just type in `a=myfunction(1)` at the command window and you get the result of `2` .

.

18.1.4: Summary of Key Points

1. In its simplest form, numerical integration involves calculating the areas of segments that make up the area under a curve,
2. MATLAB has built-in functions to perform numerical integration,
3. `Z = trapz(Y)` computes an approximation of the integral of Y using the trapezoidal method.
4. Anonymous functions are inline statements that we can define with `@(x)` ,

5. `Z = integral(fun,xmin,xmax)` numerically integrates function `fun` from `xmin` to `xmax` .

18.1.5: Footnotes

- 1 *O. N. Mathematics: 2 by J. Dobinson, Penguin Library of Technology. © 1969, (p. 184)*
- 2 *O. N. Mathematics: 2 by J. Dobinson, Penguin Library of Technology. © 1969, (p. 183)*

This page titled 18.1: Integration- Computing the Area Under a Curve is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Serhat Beyenir via source content that was edited to the style and standards of the LibreTexts platform.

- **1.6: Computing the Area Under a Curve** by Serhat Beyenir is licensed CC BY 4.0. Original source: https://cnx.org/contents/qTr2bTqz11.8:IZLgFOgb3.

18.1.1: Problem Set

Let the function y defined by $y = \cos(x)$ Plot this function over the interval [-pi,pi]. Use numerical integration techniques to estimate the integral of y over [0, pi/2] using increments of pi/10 and pi/1000.

Answer

1. Plotting x=-pi:pi/100:pi; y=cos(x); plot(x,y),title('Graph of y=cos(x)'),xlabel('x'),ylabel('y'),grid
2. Area calculation 1: >> x=0:pi/10:pi/2; >> y=cos(x); >> area1=trapz(x,y) area1 = 0.9918
3. Area calculation 2: >> x=0:pi/1000:pi/2; >> y=cos(x); >> area2=trapz(x,y) area2 = 1.0000

Let the function y defined b $y = 0.04x^2 - 2.13x + 32.58$ Plot this function over the interval [3,30]. Use numerical integration techniques to estimate the integral of y over [3,30].

Answer

1. Plotting: >> x=3:.1:30; >> y=0.04*(x.^2)-2.13.*x+32.58; >> plot(x,y), title('Graph of ... y=.04*(x^2)-2.13*x+32.58'),xlabel('x'),ylabel('y'),grid
2. Area calculation: >> area=trapz(x,y) area = 290.3868

A 2000-liter tank is full of lube oil. It is known that if lube oil is drained from the tank, the mass flow rate will decrease from the maximum when the tank level is at the highest. The following data were collected when the tank was drained.

Time [min]	**Mass Flow [kg/min]**
0	50.00
5	48.25
10	46.00
15	42.50
20	37.50
25	37.50
30	19.00
35	9.00

Data

Write a script to estimate the amount of oil drained in 35 minutes.

Answer

clc t=linspace(0,35,8) % Data entry for time [min] m=[50 48.25 46 42.5 37.5 30.5 19 9] % Data entry for mass flow [kg/min] % Calculate time intervals dt=[t(2)-t(1),t(3)-t(2),t(4)-t(3),... t(5)-t(4),t(6)-t(5),t(7)-t(6),t(8)-t(7)] % Calculate mass out dm=[0.5*(m(2)+m(1)),0.5*(m(3)+m(2)),0.5*(m(4)+m(3)),0.5*(m(5)+... m(4)),0.5*(m(6)+m(5)),0.5*(m(7)+m(6)),0.5*(m(8)+m(7))] % Calculate differential areas da=dt.*dm; % Tabulate time and mass flow [t',m'] % Tabulate time intervals, mass out and differential areas [dt',dm',da'] % Calculate the amount of oil drained [kg] in 35 minutes Oil_Drained=sum(da)

The output is:

```
ans = 0 50.0000 5.0000 48.2500 10.0000 46.0000 15.0000 42.5000 20.0000 37.5000 25.0000 30.5000 30.0000 19.0000 35.0000 9.0000 ans = 5.0000 49.1250 245.6250 5.0000 47.1250 235.6250 5.0000 44.2500 221.2500 5.0000 40.0000 200.0000 5.0000 34.0000 170.0000 5.0000 24.7500 123.7500 5.0000 14.0000 70.0000 Oil_Drained = 1.2663e+003
```

A gas is expanded in an engine cylinder, following the law $PV^{1.3}=c$. The initial pressure is 2550 kPa and the final pressure is 210 kPa. If the volume at the end of expansion is 0.75 m^3, compute the work done by the gas. [1]

Answer

```
clc disp('A gas is expanded in an engine cylinder, following the law PV^1.3=c') disp('The initial pressure is 2550 kPa and the final pressure is 210 kPa.') disp('If the volume at the end of expansion is 0.75 m3,') disp('Compute the work done by the gas.') disp(' ') % Display blank line n=1.3; P_i=2550; % Initial pressure P_f=210; % Final pressure V_f=.75; % Final volume V_i=(P_f*(V_f^n)/P_i)^(1/n); % Initial volume c=P_f*V_f^n; v=V_i:.001:V_f; % Creating a row vector for volume, v p=c./(v.^n); % Computing pressure for volume WorkDone=trapz(v,p) % Integrating p*dv.
```

The output is:

```
A gas is expanded in an engine cylinder, following the law PV^1.3=c The initial pressure is 2550 kPa and the final pressure is 210 kPa. If the volume at the end of expansion is 0.75 m3, Compute the work done by the gas. WorkDone = 409.0666
```

A force F acting on a body at a distance s from a fixed point is given by $F=3s+\frac{1}{s^2}$ Write a script to compute the work done when the body moves from the position where s=1 to that where s=10. [2]

Answer

```
clc disp('A force F acting on a body at a distance s from a fixed point is given by') disp('F=3*s+(1/(s^2)) where s is the distance in meters') disp('Compute the total work done in moving') disp('From the position where s=1 to that where s=10.') disp(' ') % Display blank line s=1:.001:10; % Creating a row vector for distance, s F=3.*s+(1./(s.^2)); % Computing Force for s WorkDone=trapz(s,F) % Integrating F*ds over 1 to 10 meters.
```

The output is:

```
A force F acting on a body at a distance s from a fixed point is given by F=3*s+(1/(s^2)) where s is the distance in meters Compute the total work done in moving From the position where s=1 to that where s=10. WorkDone = 149.4000
```

The pressure p and volume v of a given mass of gas are connected by the relation $\left(p+\frac{a}{v^2}\right)(v-b)=k$ where a, b and k are constants. Express p in terms of v, and write a script to compute the work done by the gas in expanding from an initial volume to a final volume. [3]

Test your solution with the following input:
a: 0.01
b: 0.001
The initial pressure [kPa]: 100
The initial volume [m3]: 1
The final volume [m3]: 2

Answer

```
clc % Clear screen disp('This script computes the work done by') disp('The gas in expanding from volume v1 to v2') disp(' ') % Display blank line a=input('Enter the constant a: '); b=input('Enter the constant b: '); p_i=input('Enter the initial pressure [kPa]: '); v_i=input('Enter the initial volume [m3]: '); v_f=input('Enter the final volume [m3]: '); k=(p_i+(a/(v_i^2))*(v_i-b)); % Calculating constant k v=v_i:.001:v_f; % Creating a row vector for volume p=(k./(v-b))-(a./(v.^2)); % Computing pressure for volume WorkDone=trapz(v,p); % Integrating p*dv disp(' ') % Display blank line str = ['The work
```

done by the gas in expanding from ', num2str(v_i),... ' m3 to ' num2str(v_f), ' m3 is ', num2str(WorkDone), ' kW.']; disp(str); The output is: This script computes the work done by The gas in expanding from volume v1 to v2 Enter the constant a: .01 Enter the constant b: .001 Enter the initial pressure [kPa]: 100 Enter the initial volume [m3]: 1 Enter the final volume [m3]: 2 The work done by the gas in expanding from 1 m3 to 2 m3 is 69.3667 kW.

18.1.1.1: Footnotes

- 1 *Applied Heat for Engineers by W. Embleton and L Jackson, Thomas Reed Publications. © 1999, (p. 80)*
- 2 *O. N. Mathematics: 2 by J. Dobinson, Penguin Library of Technology. © 1969, (p. 213)*
- 3 *O. N. Mathematics: 2 by J. Dobinson, Penguin Library of Technology. © 1969, (p. 212)*

This page titled 18.1.1: Problem Set is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Serhat Beyenir via source content that was edited to the style and standards of the LibreTexts platform.

- **1.6.1: Problem Set** by Serhat Beyenir is licensed CC BY 4.0. Original source: https://cnx.org/contents/qTr2bTqz11.8:IZLgFOgb3.

18.1.2: The integral() function

The integral() function is preferred over the trapz() function.

The function integral() determines a good set of x-values and uses a more sophisticated algorithm than trapz()
You need to specify a "handle" to a Matlab function, like is done for the fzero() function. The function and its handle can be specified in 2 methods:

Method A: Use a function .m file, such as this:

```
function y = y_cubic(x)
  y = x.^3 -4*x + 4
end
```

In the main script, the integral function refers to this file like this:

```
x1 = -2.0;
x2 = 1.5;
intg1 = integral(@y_cubic, x1, x2)
```

% =14.7656, which = the analytic answer

Method B: Use an "in-line" function, like this:

```
y_handle = @(x) x.^3 -4*x + 4
```

% This is also known as an "anonymous function"

```
intg1 = integral(y_handle, x1, x2)
```

% 14.7656, which = the analytic answer

You may use either a function .m file or an anonymous function definition within their script.

My opinion is that a function .m file is the clearer method.

This page titled 18.1.2: The integral() function is shared under a not declared license and was authored, remixed, and/or curated by Carey Smith.

18.2: Numerical Differentiation

18.2.1: 1-Dimensional Derivatives

In calculus, you learned that the derivative of a function is: limit of Δy/Δx as x goes to 0.

Sometimes, we only have measured data, not an analytic function, so we cannot get the derivative analytically. When the analytic derivative is unknown, we can approximate it from the data using Δy/Δx, without taking a limit. The numerical derivative at point (xn, yn) is:

(xn, yn) ≈ (yn+1 – yn-1) / (xn+1 - xn-1,)

Matlab’s function for this is named **gradient()**. There are 2 methods of calling gradient(). Both methods take 2 arguments, and the 1st argument is the vector of y-values. The 2nd argument depends on the method:

Method A: dydx_gradient = gradient(y, x) where the 2nd argument is the vector of the x values corresponding to y.

Note: Both y and x need to be inputs.
When y is the only input, gradient(y) assumes that the x spacing = 1, which gives the wrong answer by a factor of 10!

Method B. dydx_gradient = gradient(y, dx), where the 2nd argument is the spacing, dx, between the x values (dx = a single number).

Both methods give the same results when the x values are equally spaced.

✓ Example 18.2.1 Numerical Derivative of sin(x)

```
dx = 0.05*pi

x = 0: dx: 0.5*pi;

y = sin(x);

%% First, plot y vs. x
figure;
plot(x,y,'-o', 'LineWidth',2);
grid on;
title('y = sin(x)')
xlabel('x (rads)')

%% Label the (x, y) values of 3 consecutive points
text(x(5), (y(5)-0.02),['y= ',num2str(y(5),3)])
text(x(6), (y(6)-0.02),['y= ',num2str(y(6),3)])
text(x(7), (y(7)-0.02),['y= ',num2str(y(7),3)])

%% Second, plot the analytic derivative
dy_anal = cos(x); % This is the analytic derivative
figure;
%subplot(2,1,1);
plot(x,dy_anal,'s','MarkerSize',10);
grid on;
xlabel('x (rads)')

%% Approximate the derivative with the gradient function: grad()

dydx_gradientA = gradient(y, x) % Method A
dydx_gradientB = gradient(y, dx) % Method B
```

```
dydx_gradientA =
Columns 1 through 4
0.9959 0.9836 0.9472 0.8873
Columns 5 through 8
0.8057 0.7042 0.5854 0.4521
Columns 9 through 11
0.3077 0.1558 0.0784
dydx_gradientB =
Columns 1 through 4
0.9959 0.9836 0.9472 0.8873
Columns 5 through 8
0.8057 0.7042 0.5854 0.4521
Columns 9 through 11
0.3077 0.1558 0.0784
```

```
hold on;
plot(x, dydx_gradientA, '*', 'MarkerSize',10);
legend('dy\_anal\_cos','dy\_gradient')
title({'Numerical Gradient Approximates',
'the Analytic Derivative of sin(x)'})
```

Solution

.

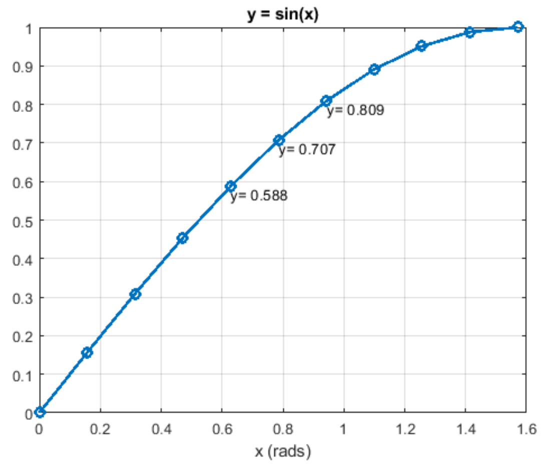


Figure 18.2.1: y = sin(x)

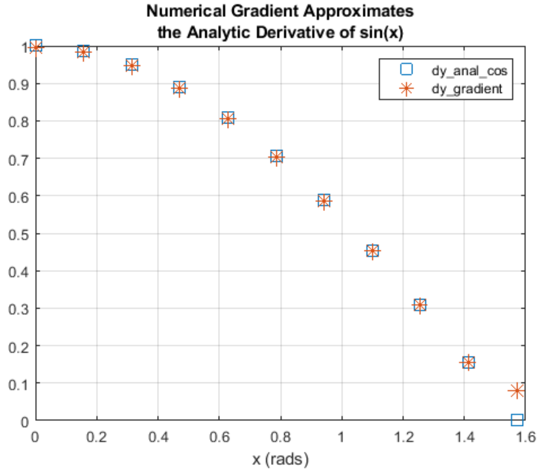


Figure 18.2.1: Derivatives of y = sin(x)

The numerical derivative is very close to the analytic values, except for the last point, because there is no point beyond the last point to improve the estimated value.

Add example text here.

.

Homework:

Exercise 18.2.2 Derivative of the normal distribution

Download the attached norm.m file which computes the normal distribution:

```
% Start a script .m file with this line:
clear all; close all; clc; format compact
% Define this vector
x = -3: 0.1 : 3;
% Compute y:
y = norm_dist(x);
%% Open a figure and plot y as a function of x:
% Turn on the grid:
%% Use gradient() to compute the derivative.
%% Then plot the gradient result on the same figure;
% If computed correctly:
% the derivative will be > 0 for x < 0
% the derivative will be = 0 for x = 0
% the derivative will be < 0 for x > 0
% Use the legend function to identify the 2 curves:
legend('norm\_dist', 'derivative')
% Add this title:
title('norm_dist and its derivative')
```

Answer

The answer is not given here.

.

18.2.2: 2 Dimensional Derivatives

Matlab's gradient function can also be used to compute the gradients of a function defined on x and y grids created by meshgrid. This example uses Matlab's built-in peaks() 3D function.

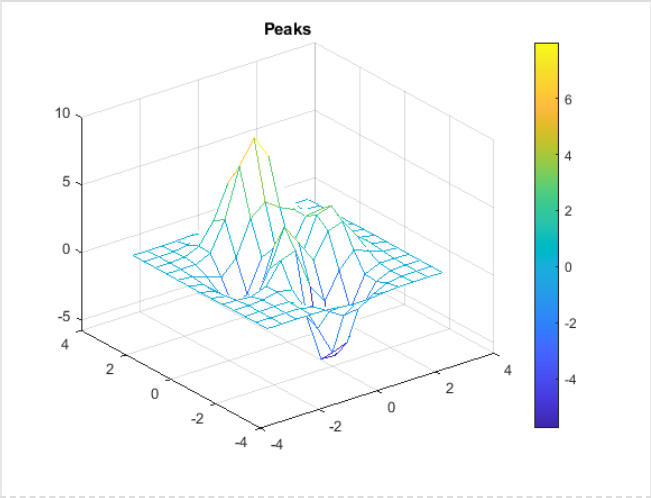
✓ Example 18.2.1

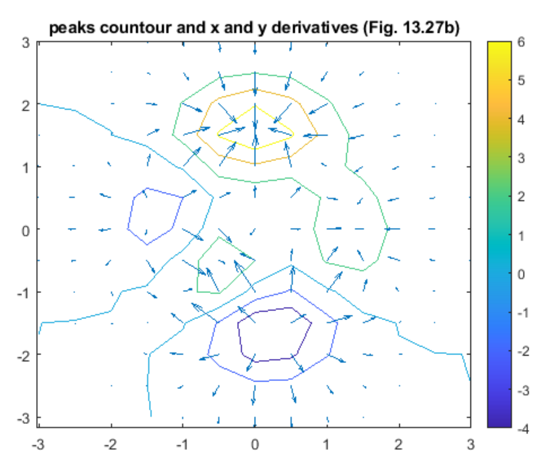
```
%% Numerical Derivatives of the Peaks 2D function
clear all; close all; clc; format compact
dx = 0.5;
x1D = (-3: dx : 3);
y1D = x1D;
[x2Da, y2Da] = meshgrid(x1D,y1D);
[x2D, y2D, z2D] = peaks(x2Da,y2Da);
figure;
mesh(x2D, y2D, z2D)
title('Peaks')

colorbar
%%
[dzdx, dzdy] = gradient(z2D, x1D, y1D);
figure;
contour(x2D, y2D, z2D);

axis equal
hold on;
% Draw the x & y cradients as arrows with the quiver plot function
quiver(x2D, y2D, dzdx, dzdy);
title('peaks countour and x and y derivatives (Fig. 13.27b)')
```

```
colorbar
```

Solution

The resulting plots are:





.

This page titled 18.2: Numerical Differentiation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

18.3: Chapter 18 Differentiation and Integration Videos in English and Espanol

18.3.1: Videos in English:

astering MATLAB Function Handle
rtan Professor
Function Handles
and Function Functions
Watch on YouTube

.

18.3.2: Videos in Spanish

egrales con Matlab: int vs integral
ses Debbie (Dra. D)
con
Matlab
Comandos
int
Watch on YouTube

La primera parte e para symbolica toolbox.

Comence a 6:05 para MATLAB integral()

.

This page titled 18.3: Chapter 18 Differentiation and Integration Videos in English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

CHAPTER OVERVIEW

19: Ordinary Differential Equations

In the previous chapter, we found the equilibrium point where a duck would float on water. This kind of problem is called $static$ because it does not move. This chapter introduces $dynamic$ problems, which involve things that change over time.

Also, you'll learn about a mathematical tool for describing physical systems, *differential equations*, and two computational tools for solving them, Euler's method and `ode45`.

But first I have a quick suggestion about organizing code in files.

- 19.1: Functions and Files
- 19.2: Differential Equations
- 19.3: Euler’s Method
- 19.4: Implementing Euler’s Method
- 19.5: Solving ODEs with ode45
  - 19.5.1: ODE45 Examples and Exercises
- 19.6: Time Dependence
- 19.7: What Could Go Wrong?
- 19.8: Labeling Axes
- 19.9: Chapter Review
- 19.10: Chapter 19 ODE Videos in English and Espanol
- 19.11: Exercise

This page titled 19: Ordinary Differential Equations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

19.1: Functions and Files

So far we've only put one function in each file. It's also possible to put more than one function in a file, but only the first one, the *top-level function*, can be called from the Command Window. The other *helper functions* can be called from anywhere inside the file, but not from the Command Window or any other file.

Keeping multiple functions in one file is convenient, but it makes debugging difficult because you can't call helper functions from the Command Window.

To help with this problem, I often use the top-level function to develop and test my helper functions. For example, to write a program for Section 7.2, I would create a file named *duck.m* and start with a top-level function named `duck` that takes no input variables and returns no output value.

Then I would write a function named `error_func` to evaluate the error function for `fzero`. To test `error_func`, I would call it from `duck` and then call `duck` from the Command Window.

Here's what my first draft might look like:

```
function res = duck()
    error = error_func(10)
end

function res = error_func(d)
    rho = 0.3;      % density in g / cm^3
    r = 10;         % radius in cm
    res = d;
end
```

This program is not complete, but it is enough code to test. Once this program is working, I would finish writing `error_func`. And once I'd finished and tested `error_func`, I would modify `duck` to use `fzero`.

This problem might only require two functions, but if there were more, I could write and test them one at a time and then combine them into a working program.

Now, let's get back to differential equations.

This page titled 19.1: Functions and Files is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.1: Functions and Files** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

19.2: Differential Equations

A *differential equation (DE)* is an equation that describes the derivatives of an unknown function. “Solving a DE” means finding a function whose derivatives satisfy the equation.

For example, suppose we would like to predict the population of yeast growing in a nutrient solution. Assume that we know the initial population is 5 billion yeast cells. When yeast grow in particularly yeast-friendly conditions, the rate of growth at any point in time is proportional to the current population. If we define $y(t)$ to be the population at a time t, we can write the following equation for the rate of growth:

$$\frac{dy}{dt}(t) = ay(t)$$

where $\frac{dy}{dt}(t)$ is the derivative of $y(t)$ and a is a constant that characterizes how quickly the population grows. This equation is *differential* because it relates a function to one of its derivatives.

It is an *ordinary differential equation (ODE)* because all the derivatives involved are taken with respect to the same variable. If it related derivatives with respect to different variables (partial derivatives), it would be a *partial differential equation (PDE)*.

This equation is *first order* because it involves only first derivatives. If it involved second derivatives, it would be second order, and so on.

Lastly, it’s *linear* because each term involves t, y, or dy/dt raised to the first power; if any of the terms involved products or powers of t, y, or dy/dt it would be *nonlinear*.

Now suppose we want to predict the yeast population in the future. We can do that using Euler’s method.

This page titled 19.2: Differential Equations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.2: Differential Equations** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

19.3: Euler's Method

Here's a test to see if you're as smart as Leonhard Euler. Let's say you arrive at time (t) and measure the current population (y) and the rate of change (r). What do you think the population will be after some period of time Δt has elapsed?

If you said $y + r\Delta t$, congratulations! You just invented Euler's method.

This estimate is based on the assumption that r is constant, but in general it's not, so we only expect the estimate to be good if r changes slowly and Δt is small.

What if we want to make a prediction when Δt is large? One option is to break Δt into smaller pieces, called *time steps*. Then we can use the following equations to get from one time step to the next:

$$\begin{aligned} T_{i+1} &= T_i + dt \\ Y_{i+1} &= Y_i + \frac{df}{dt}(t)\, dt \end{aligned} \tag{19.3.1}$$

Here, T_i is a sequence of times where we estimate the value of y, and Y_i is the sequence of estimates. For each index i, Y_i is an estimate of $y(T_i)$.

If the rate doesn't change too fast and the time step isn't too big, Euler's method is accurate enough for most purposes.

This page titled 19.3: Euler's Method is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.3: Euler's Method** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

19.4: Implementing Euler’s Method

As an example we’ll use Euler’s method to solve the equation from page ,

$$\frac{dy}{dt}(t) = ay(t)$$

with the initial condition $y(0) = 5$ billion cells and the growth parameter $a = 0.2$ per hour.

As a first step, create a file named *euler.m* with a top-level function and a helper function:

```
function res = euler()
    T(1) = 0;
    Y(1) = 5;
    r = rate_func(T(1), Y(1))
end

function res = rate_func(t, y)
   a = 0.2;
   dydt = a * y;
   res = dydt;
end
```

In `euler` we initialize the initial conditions and then call `rate_func`, so called because it computes the rate of growth in the population.

After testing these functions, we can add code to `euler` to compute these difference equations:

$$\begin{aligned} T_{i+1} &= \quad T_i + \Delta t \\ Y_{i+1} &= \quad Y_i + r\Delta t \end{aligned} \tag{19.4.1}$$

where r is the rate of growth computed by `rate_func`. Listing 9.1 has the code we need:

Listing 9.1: A function implementing Euler's method

```
function res = euler()
    T(1) = 0;
    Y(1) = 5;
    dt = 0.1;

    for i=1:40
        r = rate_func(T(i), Y(i));
        T(i+1) = T(i) + dt;
        Y(i+1) = Y(i) + r * dt;
    end
    plot(T, Y)
end
```

Before the loop, we create two vectors, `T` and `Y`, and set the first element of each with the initial conditions; `dt`, which is the size of the time steps, is 0.1 hours.

Inside the loop, we compute the growth rate based on the current time, `T(i)`, and population, `Y(i)`. You might notice that the rate depends only on population, but we pass time as an input variable anyway, for reasons you’ll see soon.

After computing the growth rate, we add an element both `T` and `Y`. Then, when the loop exits, we plot `Y` as a function of `T`.

If you run the code, you should get a plot of population over time, as shown in Figure 9.1.

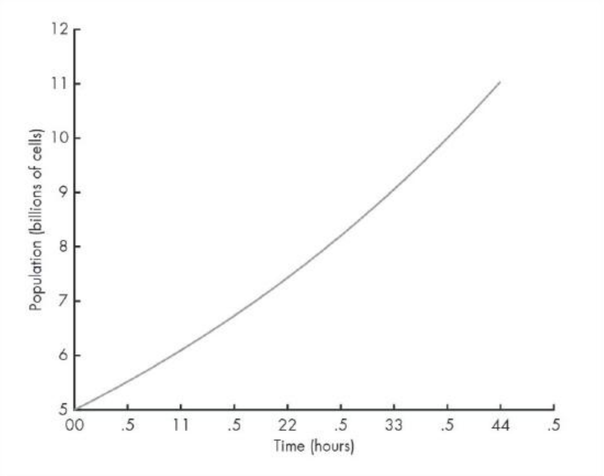


Figure 9.1: Solution to a simple differential equation by Euler’s method

As you can see, the population doubles in a little less than 4 hours.

This page titled 19.4: Implementing Euler’s Method is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.4: Implementing Euler’s Method** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

SECTION OVERVIEW

19.5: Solving ODEs with ode45

A limitation of Euler's method is that it assumes that the derivative is constant between time steps, and that's not generally true. Fortunately, there are better methods that estimate the derivative between time steps, and they are much more accurate.

MATLAB provides a function called `ode45` that implements one of these methods. In this section I'll explain how to use it; you can read more about how it works in "" on page .

In order to use `ode45` , you have to write a function that evaluates dy/dt as a function of t and y. Fortunately, we already have one, called `rate_func` :

```
function res = rate_func(t, y)
   a = 0.2;
   dydt = a * y;
   res = dydt;
end
```

We can call `ode45` from the Command Window like this:

```
[T, Y] = ode45(@rate_func, [0, 4], 5);
plot(T, Y)
```

The first argument is a function handle, as we saw in Chapter . The second argument is the time interval where we want to evaluate the solution; in this case the interval is from $t = 0$ to $t = 4$ hours. The third argument is the initial population, 5 billion cells.

The `ode45` function is the first function we've seen that returns *two* output variables. In order to store them, we have to assign them to two variables, `T` and `Y` . Figure 9.2 shows the results.

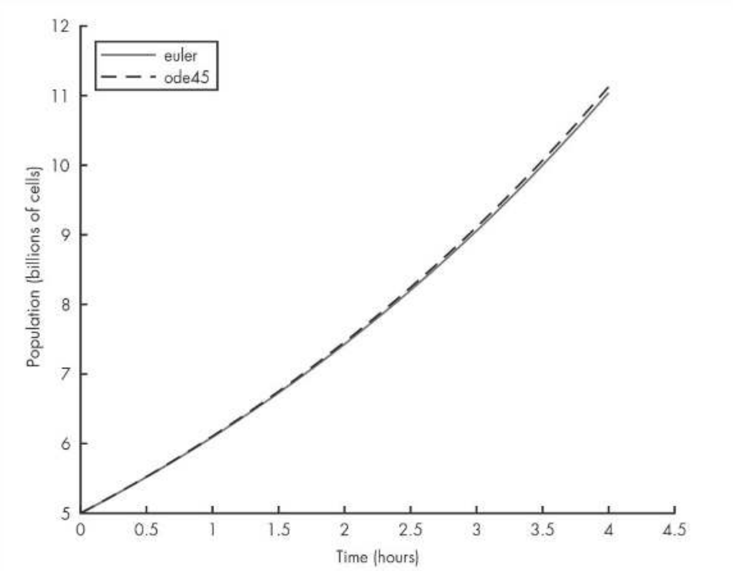


Figure 9.2: Solutions to a simple differential equation using Euler's method and `ode45`

The solid line is the estimate we computed with Euler's method; the dashed line is the solution from `ode45` .

For the first 2–3 hours, the two solutions are visually indistinguishable. During the last hour, they diverge slightly; at 4 hours, the difference is less than 1 percent.

For many purposes, the difference between Euler’s method and `ode45` is the least of our worries. In this example, we probably don’t know the initial population with perfect accuracy or the growth constant, `a`. Also, the assumption that the growth rate only depends on population is probably not true. Any of these modeling errors could be bigger than 1 percent.

However, for some problems, Euler’s method can be off by a lot more than 1 percent. In those cases `ode45` is almost always more accurate, for two reasons: first, it computes the rate function several times per time step; second, if the time step is too big, `ode45` can detect the problem and shrink the time step. For more details, see “How ode45 Works” on page .

19.5.1: ODE45 Examples and Exercises

This page titled 19.5: Solving ODEs with ode45 is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

SECTION OVERVIEW

19.5.1: ODE45 Examples and Exercises

If you have taken a differential equations course, then you learned techniques to solve a variety of differential equations analytically.
Other ODE's cannot be solved analytically. They can only be solved numerically.

ode45() requires a differential equation function to be coded in a particular format. This function can be implemented in 3 ways in MATLAB and 2 ways in Octave.

1. The ODE function can be a coded as its own file. This is straight-forward. It works in both MATLAB and Octave. It is illustrated in the 2nd example.
2. An "anonymous function" (an in-line function). It can be used when the function is a single line of code. This can work for a 1st order ode, but not easily for a 2nd order ode. using an An anonymous function is not illustrated here.
3. The ODE function can be placed at the bottom of the script file. This is called a subfunction. It works in MATLAB, but not in the current version of Octave. (However Octave allows a subfunction within a function file, but not in a script file.) It is illustrated in the 1st example.
4. The simplest way to use ODE45() is specify 3 general parameters in the argument list and put the function specific parameters in the function file or anonymous specification. The general parameters are:
    1. The function handle
    2. The time interval
    3. The initial vector of x and its derivatives.
5. The function specific parameters can be specified in the function call, as shown in the last example below. This takes more care to be done correctly.

In previous chapters in this book, we have generally used separate files for functions used by MATLAB functions such as fzero() and integral(). I recommend creating a separate file for function that could be used multiple times with different inputs; this is a general purpose function. However, for the ode() function, the user codes a function with specific parameters defined within the function (not general purpose), so it makes sense to define it as a subfunction at the bottom of the script file (if you are using MATLAB).

The 1st example of using the ode45() function is a 1st order ODE; it can be easily solved analytically. The example solves it with ode45(); the solution can be verified vs. the analytical solution.

The 2nd example of using the ode45() function is a 2nd order ODE; it can be solved analytically only with difficulty.

✓ Example 19.5.1.1 ODE of a Parabola

This example if of a parabola as a function of time. The code for this example illustrates using a sub-function within the test file.

The equation equation for this parabola is

y(t) = 2*t^2 - 0.5

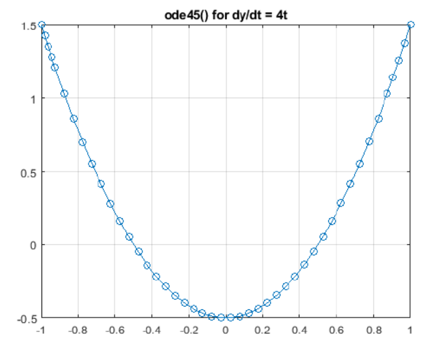
```
% dydt_4t_ODE_example.m
% This example uses a sub-function within the test file.
% The equation equation for this parabola is y(t) = 2*t^2 -4

% The syntax of ode45 is:
% [T_OUT, Y_OUT] = ode45(@ODEFUN, TSPAN, Y0)
% @ODEFUN is the handle to the ode function.
TSPAN = [-1.0, 1.0]; % s
```

```
% TSPAN is a 2-element vector = [T0, TFINAL]
% specifying the time interval
Y0 = 1.5; % the initial value of y(t)
[tout, yout] = ode45(@dy4_ODE, TSPAN, Y0);
% Note: Instead of the user specifying a time vector,
% ode45 computes a time vector with appropriate time increments.

% Plot the function computed by ode45
figure;
plot(tout, yout, '-o')
title('ode45() for dy/dt = 4t')
grid on;

function dydt = dy4_ODE(t,y)
dydt = 4*t;
end
```

The plot is shown in the solution.

Solution

A plot of the solution.



.

Example 19.5.1.2 Damped spring example

This example uses a separate function file.

The 2nd-order differential equation for an under-damped spring oscillator is:

$$m\frac{d^2x}{dt^2}+c\frac{dx}{dt}+kx=0$$

The constants for this example are:

m = 2
c = 0.4
k = 4

The differential equation and the constants are defined in the function file "ODE_Damped_Spring.m"

For a 2nd order differential equation, **ODE45 requires that the function file has a column vector of 2 inputs and a column vector of 2 outputs.**

The input vector, x, has the position and it's 1st derivative:

x(1) = x = the position
x(2) = dx/dt = the 1st derivative of x(1)

The output vector is the derivative of the input vector:

xdot(1) = dx/dt

xdot(2) = d^2x/dt^2 = the derivative of dx/dt

$$\frac{d^2x}{dt^2} = -\frac{1}{m}(c\frac{dx}{dt} + kx)$$

The code for the function is given here and is also attached as ODE_Damped_Spring.m

```
function xdot = ODE_Damped_Spring(t, x)
% This models the oscillation of a damped spring
% Inputs:
% t = time is not used, since this is a time-independent model
% x = a column vector with 2 elements:
% x(1) = x = the position
% x(2) = dx/dt = the 1st derivative of x
% Output
% xdot = a column vector with 2 elements:
% xdot(1) = 1st derivative of x = x(2)
% xdot(2) = 2nd derivative of x from this equation:
% m(d^2x/dt^2) + c(dx/dt) + kx = 0
% Constants of the equations
m = 2;
c = 0.4;
k = 4;
xdot = zeros(2,1); % Initialize the output
xdot(1) = x(2); % 1st derivative
xdot(2) = -(1/m)*(c*x(2) + k*x(1)); % 2nd derivative
```

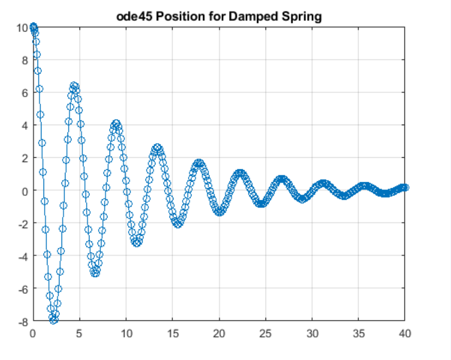
The test program sets the "initial conditions" (the starting position and derivative), calls ODE45 with a handle to the diferential equation function, and plots the result. This is given here and is also attached as ODE_Damped_Spring_test.m.

```
%% ODE_Damped_Spring_test
% m(d^2x/dt^2) + c(dx/dt) + kx = 0
% m = 2;
% c = 0.4;
% k = 4;
% To compute the solution to this equation,
% we need a function: ODE_Damped_Spring.m
% The x input needs to be a column vector.
x0 = [10;  % Initial values of x
       0]; % Initial values the 1st derivative x
Time_Span = [0, 40.0]; % (s)
[t_ex2, x_ex2] = ode45(@ODE_Damped_Spring, Time_Span, x0);

figure;
plot(t_ex2, x_ex2(:,1), 'o-')
title('ode45 Position for Damped Spring')
grid on;
```

The resulting plot is shown in the solution.

Solution

The resulting plot is shown here.



.

Example 19.5.1.2 Passing function parameters thru the ODE45 call

This is the differential equation function file is shown here. Note that in addition to the required x and t inputs, it has 3 function specif parameters: m, c, &

```
function dxdt = damped_spring1(t, x, m, c, k)
dxdt = zeros(2, 1);
dxdt(1) = x(2);
dxdt(2) = (-c*x(2) - k*x(1)) / m;
end
```

The main script that includes the ode45() function call is this:

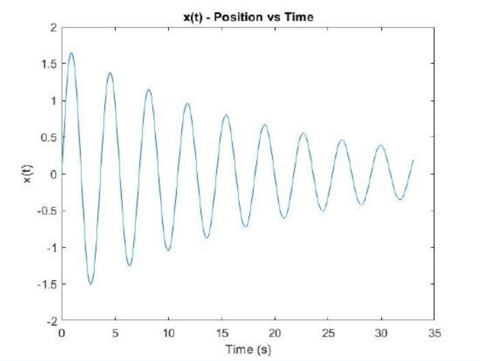
```
m = 3;
c= 0.3;
k= 9;

time_span = [0 33];
xx0 = [0; 3];

[t,xx] = ode45(@(t, x) damped_spring1(t, x, m, c, k), time_span, xx0);

figure
plot(t, xx(:,1))
title ('x(t) - Position vs Time')
xlabel('Time (s)')
ylabel('x(t)')

figure
plot(t, xx(:,2))
title('dx/dt = Velocity vs Time')
xlabel('Time (s)')
ylabel('dx/dt')
```

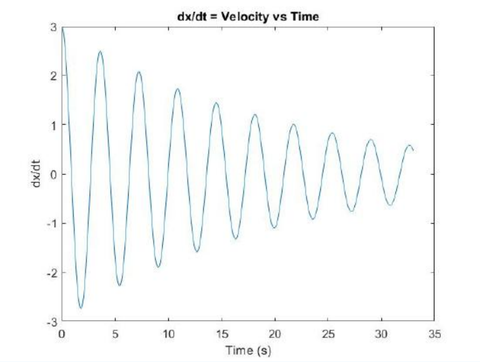
In the standard form of calling ode45(), one does not need to specify t and x.

But in this advanced form, t and x need to be specified, followed by the function-specific parameters:

[t,xx] = ode45(@(t, x) damped_spring1(t, x, m, c, k), time_span, xx0);

Solution





.

Exercise 19.5.1.1 Logistic ODE Equation Assignment

Background:

The Logistic ODE equation is:

dP/dt = r*P*(1 - P/K)

where:

P = population

r = growth rate when the population is small
K = carrying capacity (max. value of P)

This a modification of the version of the exponential population growth ODE which assumes that the population is changing at a constant rate.

When the population population is small, P/K is small, then dP/dt is approximately = r*P. This means that the growth rate is proportional to the population and P grows exponentially.

When the population population becomes close to K, then P/K is near 1, so dP/dt is approximately = 0 and P is approximately constant.

Assignment:

Create an m-file script that computes the solution to the logistic differential equation.

Start with this line:

```
clear all; close all; clc; format compact
```

(1 pt) Set these parameters:

```
tspan = [1,100] % [timeBegin, timeEnd] = the time span for the computation.
```

```
P0 = 10 % Initial population
```

(5 pts) Write the 1 line of code that calls the **ode45()** solver function with its 3 inputs:

@dPdt (the function handle), tspan, and P0

The outputs are [t, P]

(2 pts) Open a figure and plot P vs. t with this code:

```
figure;
plot(t, P);
title('Logistic Equation')
xlabel('time')
ylabel('Population')
grid on;
```

(2 pts) Put the following code that computes the derivative of P as a local function at the bottom of your m-file:

%% Local function

```
% Ordinary Differential Equation
function dPdt = logistic_ode(t, P)
r = 0.065; % growth rate
K = 38; % max value of P
dPdt = r*P*(1 - P/K);
end
```

If you want to learn more about the Logistic ODE, visit these links:

https://mathworld.wolfram.com/LogisticEquation.html (Links to an external site.)
https://www.matlab-monkey.com/ODE/logisticV1/logisticV1Pub.html (Links to an external site.)

Answer

The solution is not given here. Your plot should show the population increasing until it gets near to the maximum value, K, then the curve becomes nearly horizontal.

.

This page titled 19.5.1: ODE45 Examples and Exercises is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

19.6: Time Dependence

Looking at `rate_func` in the previous section, you might notice that it takes `t` as an input variable but doesn't use it. That's because the growth rate does not depend on time—bacteria don't know what time it is.

But rats do. Or, at least, they know what season it is. Suppose that the growth rate for rats depends on the current population *and* the availability of food, which varies over the course of the year. The differential equation might be something like

$$\frac{dy}{dt}(t) = ay(t)\,(1 - \cos(\omega t))$$

where t is time in days and $y(t)$ is the population at time t. Because the growth rate depends on time, this differential equation is *time dependent*.

The variables a and ω are *parameters*, which are values that quantify a physical aspect of the scenario. Parameters are often constants, but in some models they vary in time.

In this example, a characterizes the reproductive rate per day, and ω is the frequency of a periodic function that describes the effect of varying food supply on reproduction.

We'll use the values $a = 0.002$ and $\omega = 2\pi/365$ (one cycle per year). The growth rate is lowest at $t = 0$, on January 1, and highest at $t = 365/2$, on June 30.

Now we can write a function that evaluates the growth rate:

```
function res = rate_func(t, y)
    a = 0.002;
    omega = 2*pi / 365;
    res = a * y * (1 - cos(omega * t));
end
```

To test this function, I put it in a file called *rats.m* with a top-level function called `rats`:

```
function res = rats()
    t = 365/2;
    y = 1000;
    res = rate_func(t, y);
end
```

The top-level function assumes, for purposes of testing, that there are 1000 rats at $t = 365/2$ (June 30) and computes the growth rate under those conditions.

We can run the top-level function like this:

```
>> r = rats

r = 4
```

Under these conditions, the growth rate is 4 new rats per day.

Now that we've tested `rate_func`, we can use `ode45` to solve the differential equation. Here's how to call it from the top-level function in *rats.m*:

```
[T, Y] = ode45(@rate_func, [0, 365], 1000)
plot(T, Y)
```

The first argument is a function handle, again. The second argument is the interval we are interested in, a duration of one year, expressed in units of days. The third argument is the initial population, $y(0) = 1000$.

Figure 9.3 shows the results.

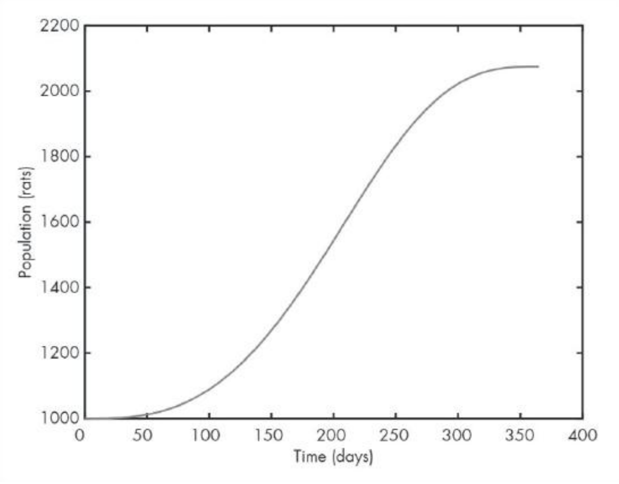


Figure 9.3: Solutions to a simple differential equation by Euler's method and `ode45`

The population grows slowly during the winter, quickly during the summer, and then slowly again in the fall.

To see the population at the end of the year, you can display the last element of `Y` :

```
Y(end)
2.0751e+03
```

That's a little more than 2000 rats, so the population roughly doubles in one year.

The index here is `end` , which is a special word in MATLAB that means "the index of the last element." You can use it in an expression, so `Y(end - 1)` is the second-to-last element of `Y` .

This page titled 19.6: Time Dependence is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.6: Time Dependence** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

19.7: What Could Go Wrong?

Don't forget the `@` on the function handle. If you leave it out, like:

```
[T, Y] = ode45(rate_func, [0, 365], 1000)
```

MATLAB treats the first argument as a function call and calls `rate_func` without providing arguments. Then you get an error message:

```
Not enough input arguments.

Error in rats>rate_func (line 18)
    res = a * y * (1 - cos(omega * t));

Error in rats (line 6)
    [T, Y] = ode45(rate_func, [0, 365], 1000);
```

Also, the rate function you write has to take two input variables, `t` and `y`, in that order, and return one output variable, `res`.

If you're working with a rate function like:

$$\frac{dy}{dt}(t) = ay(t)$$

you might be tempted to write this:

```
function res = rate_func(y)        % WRONG
    a = 0.002;
    res = a * y;
end
```

But that would be wrong. So very wrong. Why? Because when `ode45` calls `rate_func`, it provides two arguments. If you only take one input variable, you'll get an error. So you have to write a function that takes `t` as an input variable, even if you don't use it:

```
function res = rate_func(t, y)     % RIGHT
    a = 0.002;
    res = a * y;
end
```

Another common error is to write a function that doesn't make an assignment to the output variable. If you write something like:

```
function res = rate_func(t, y)
    a = 0.002;
    omega = 2*pi / 365;
    r = a * y * (1 - cos(omega * t));    % WRONG
end
```

and then call it from `ode45`, you get

```
Output argument "res" (and maybe others) not assigned during call
to "rate_func".
```

I hope these warnings save you some time debugging.

This page titled 19.7: What Could Go Wrong? is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.7: What Could Go Wrong?** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

19.8: Labeling Axes

The plots in this chapter have labels on the axes, and one of them has a legend, but I didn't show you how to do that. Let's do it now.

The functions to label the axes are `xlabel` and `ylabel`:

```
xlabel('Time (hours)')
ylabel('Population (billions of cells)')
```

The function to generate a legend is `legend`:

```
legend('euler', 'ode45')
```

The arguments are the labels for the lines, in the order they were drawn. Usually the legend is in the upper-right corner, but you can move it by providing an optional argument called `Location`:

```
legend('euler', 'ode45', 'Location', 'northwest')
```

Finally, save the figures using `saveas`:

```
saveas(gcf, 'runge.eps', 'epsc')
```

The first argument is the figure we want to save; `gcf` is a MATLAB command that stands for "get current figure," which is the figure we just drew. The second argument is the filename. The extension specifies the format we want, which is Encapsulated PostScript (*.eps*). The third argument tells MATLAB what driver to use. The details aren't important, but `epsc` generates figures in color.

This page titled 19.8: Labeling Axes is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.8: Labeling Axes** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

19.9: Chapter Review

This chapter introduced *differential equations (DE)*, which are equations that describe the derivatives of an unknown function. In an *ordinary differential equation (ODE)*, all derivatives are taken with respect to the same variable, as opposed to a *partial differential equation (PDE)*, which includes derivatives with respect to more than one variable.

A *first-order DE* includes only first derivatives, and a *linear DE* includes no products or powers of the function and its derivatives. A differential equation is *time dependent* if the rate function depends on time.

When we solve a differential equation numerically, the *time step* is the interval in time between successive elements of the solution. A *parameter* is a value that appears in a model to quantify some physical aspect of the scenario being modeled.

Until now we have only put one function in each M-file, but in this chapter we wrote a *top-level function*, which is the first function in an M-file, and a *helper function*, which is any function in an M-file that is not the top-level function.

In the next chapter, we'll solve *systems* of ODEs, which are used to describe physical systems with multiple parts that interact. But first, here's an exercise where you can apply what you've learned so far.

This page titled 19.9: Chapter Review is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.9: Chapter Review** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

19.10: Chapter 19 ODE Videos in English and Espanol

19.10.1: English

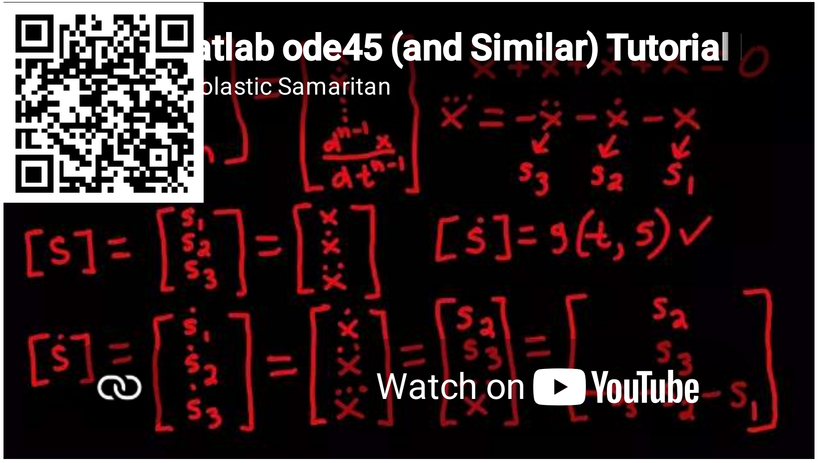
https://www.mathworks.com/videos/sol...239476787.html

w to Use Built-In ODE Solvers in M
TLAB
TO
UILT-IN
ODE SOLVERS IN
MATLAB
Watch on YouTube

.

This next is a 49 minute more detailed video. It is the 1st in a sequence of videos. The others are more advanced topics.

https://www.youtube.com/watch?v=hR9O7sVOArE&t=1s


atlab ode45 (and Similar) Tutorial
plastic Samaritan
Watch on YouTube

.

19.10.2: Espanol

uaciones Diferenciales Ordinarias
io A. Castaño Giraldo
Algoritmos
ode
31
Watch on YouTube

.

19.10.3: Links to Advanced Material

For advanced material, see in Allen Downey's book at either of these links:

https://greenteapress.com/wp/physical-modeling-in-matlab/

Physical Modeling in MATLAB (Downey)

.

This page titled 19.10: Chapter 19 ODE Videos in English and Espanol is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

19.11: Exercise

Before you go on, you might want to work on the following exercise.

Exercise 9.1

Suppose that you're given an 8-ounce cup of coffee at 90 °C. You have learned from bitter experience that the hottest coffee you can drink comfortably is 60 °C.

If the temperature of the coffee drops by 0.7 °C during the first minute, how long will you have to wait to drink your coffee?

You can answer this question with Newton's Law of Cooling (see https://greenteapress.com/matlab/newton):

$$\frac{dy}{dt}(t) = -k(y(t) - e)$$

where $y(t)$ is the temperature of the coffee at time t, e is the temperature of the environment, and k is a parameter that characterizes the rate of heat transfer from the coffee to the environment.

Let's assume that e is 20 °C and constant; that is, the coffee does not warm up the room.

Let's also assume k is constant. In that case, we can estimate it based on the information we have. If the temperature drops 0.7 °C during the first minute, when the coffee is 90 °C, we can write

$$-0.7 = -k(90 - 20)$$

Solving this equation yields $k = 0.01$.

Here are some suggestions for getting started:

1. Create a file named *coffee.m* and write a function called `coffee` that takes no input variables. Put a simple statement like `x = 5` in the body of the function and invoke `coffee` from the Command Window.
2. Add a helper function called `rate_func` that takes `t` and `y` and computes dy/dt. In this case, `rate_func` does not actually depend on t; nevertheless, your function has to take t as the first input variable in order to work with `ode45`.
3. Test your function by adding a line like `rate_func(0, 90)` to `coffee`, then call `coffee` from the Command Window. Confirm that the initial rate is -0.7 °C/min.
4. Once you get `rate_func` working, modify `coffee` to use `ode45` to compute the temperature of the coffee for 60 minutes. Confirm that the coffee cools quickly at first, then cools more slowly, and reaches room temperature after about an hour.
5. Plot the results and estimate the time when the temperature reaches 60 °C.

This page titled 19.11: Exercise is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Allen B. Downey (Green Tea Press) via source content that was edited to the style and standards of the LibreTexts platform.

- **9.10: Exercise** by Allen B. Downey is licensed CC BY-NC 4.0. Original source: https://github.com/AllenDowney/PhysicalModelingInMatlab.

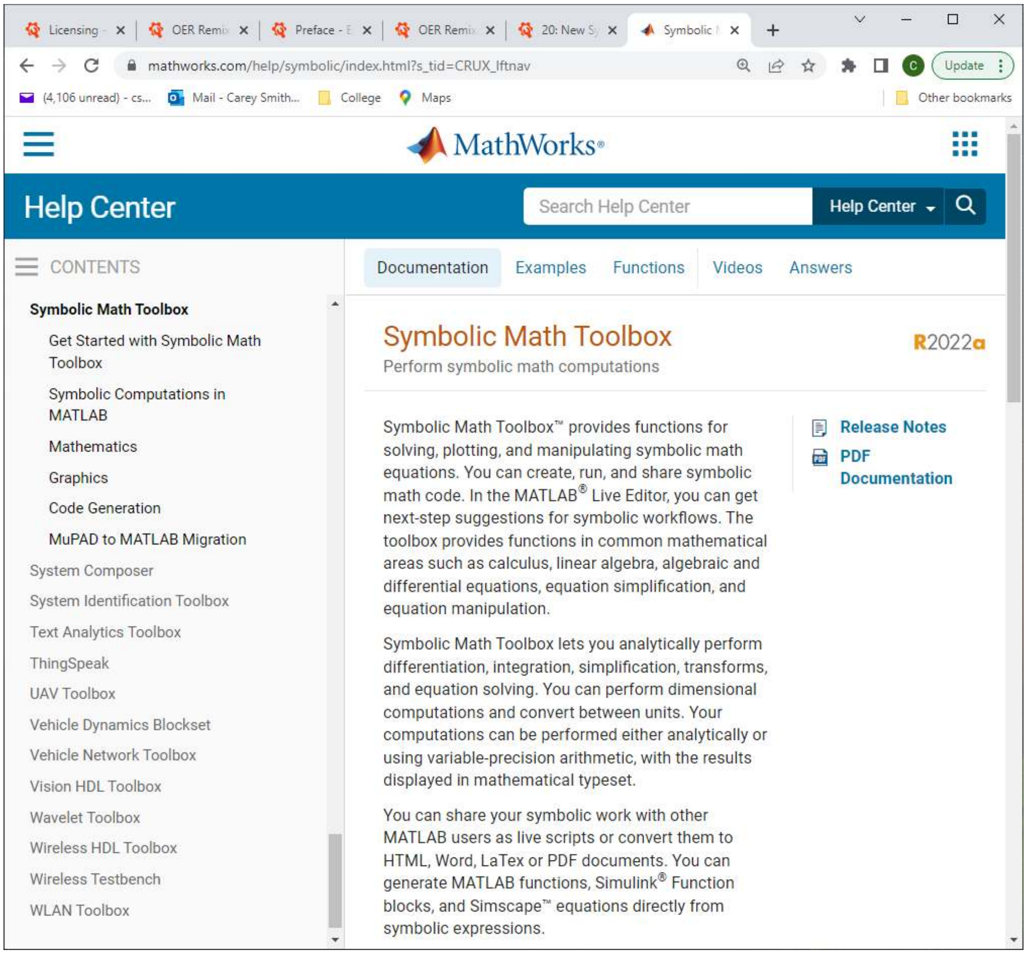
CHAPTER OVERVIEW

20: Symbolic Processing

MATLAB

A series of tutorials for MATALB's Symbolic Math Toolbox are found at this link:

https://www.mathworks.com/help/symbolic/index.html?s_tid=CRUX_lftnav


Licensing
OER Remi
Preface - E
OER Remi
20: New Sy
Symbolic
mathworks.com/help/symbolic/index.html?s_tid=CRUX_lftnav
Update
(4,106 unread) - cs...
Mail - Carey Smith...
College
Maps
Other bookmarks
MathWorks®
Help Center
Search Help Center
Help Center
CONTENTS
Documentation
Examples
Functions
Videos
Answers
Symbolic Math Toolbox
Get Started with Symbolic Math Toolbox
Symbolic Computations in MATLAB
Mathematics
Graphics
Code Generation
MuPAD to MATLAB Migration
System Composer
System Identification Toolbox
Text Analytics Toolbox
ThingSpeak
UAV Toolbox
Vehicle Dynamics Blockset
Vehicle Network Toolbox
Vision HDL Toolbox
Wavelet Toolbox
Wireless HDL Toolbox
Wireless Testbench
WLAN Toolbox
Release Notes
PDF Documentation

Symbolic Math Toolbox

Perform symbolic math computations

R2022a

Symbolic Math Toolbox™ provides functions for solving, plotting, and manipulating symbolic math equations. You can create, run, and share symbolic math code. In the MATLAB® Live Editor, you can get next-step suggestions for symbolic workflows. The toolbox provides functions in common mathematical areas such as calculus, linear algebra, algebraic and differential equations, equation simplification, and equation manipulation.

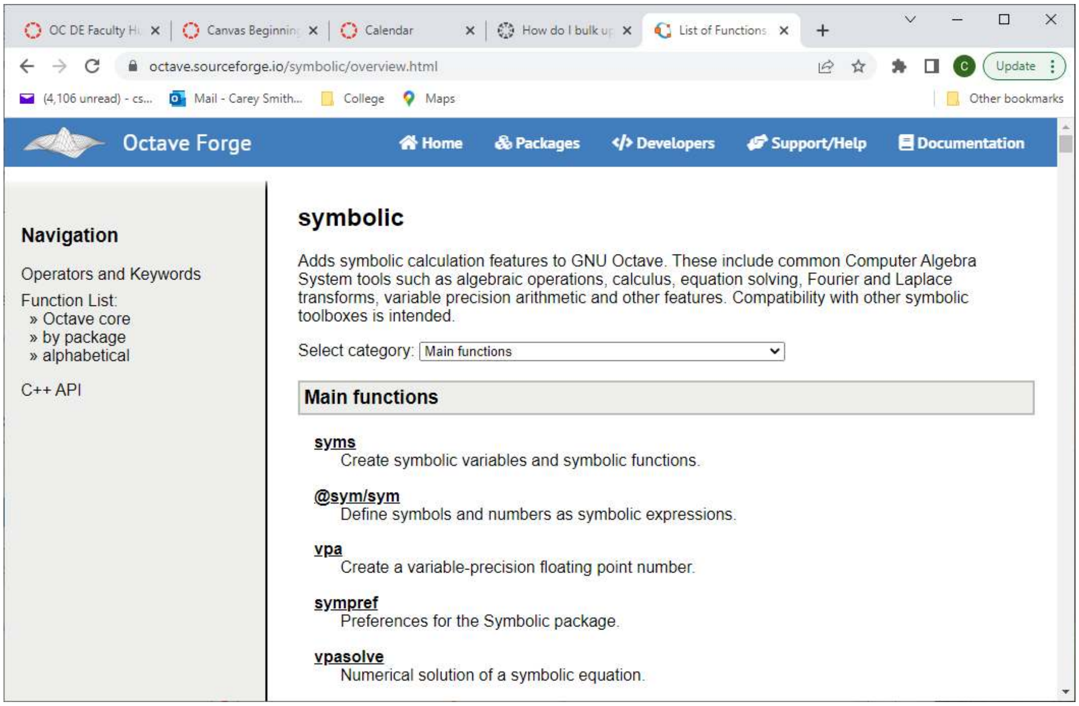
Symbolic Math Toolbox lets you analytically perform differentiation, integration, simplification, transforms, and equation solving. You can perform dimensional computations and convert between units. Your computations can be performed either analytically or using variable-precision arithmetic, with the results displayed in mathematical typeset.

You can share your symbolic work with other MATLAB users as live scripts or convert them to HTML, Word, LaTex or PDF documents. You can generate MATLAB functions, Simulink® Function blocks, and Simscape™ equations directly from symbolic expressions.

Octave

Octave's symbolic package can be obtained here:

https://octave.sourceforge.io/symbolic/


OC DE Faculty H
Canvas Beginning
Calendar
How do I bulk up
List of Functions
octave.sourceforge.io/symbolic/overview.html
(4,106 unread) - cs...
Mail - Carey Smith...
College
Maps
Other bookmarks
Update
Octave Forge
Home
Packages
Developers
Support/Help
Documentation

Navigation

Operators and Keywords

Function List:
- » Octave core
- » by package
- » alphabetical

C++ API

symbolic

Adds symbolic calculation features to GNU Octave. These include common Computer Algebra System tools such as algebraic operations, calculus, equation solving, Fourier and Laplace transforms, variable precision arithmetic and other features. Compatibility with other symbolic toolboxes is intended.

Select category: Main functions

Main functions

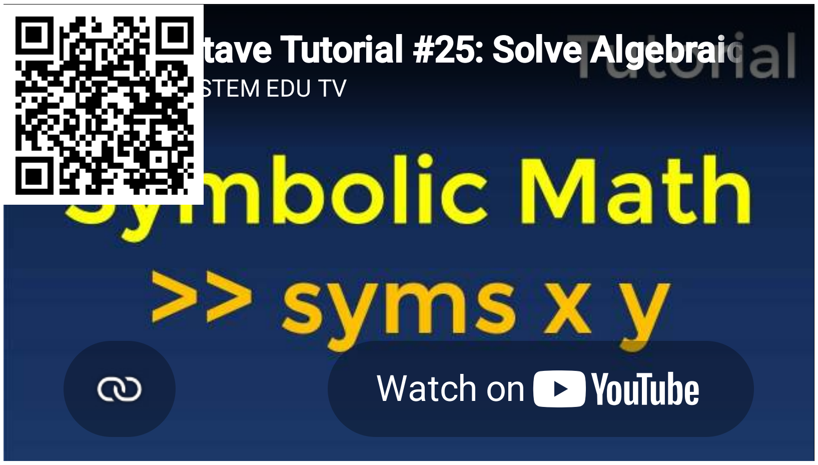
syms
Create symbolic variables and symbolic functions.

@sym/sym
Define symbols and numbers as symbolic expressions.

vpa
Create a variable-precision floating point number.

sympref
Preferences for the Symbolic package.

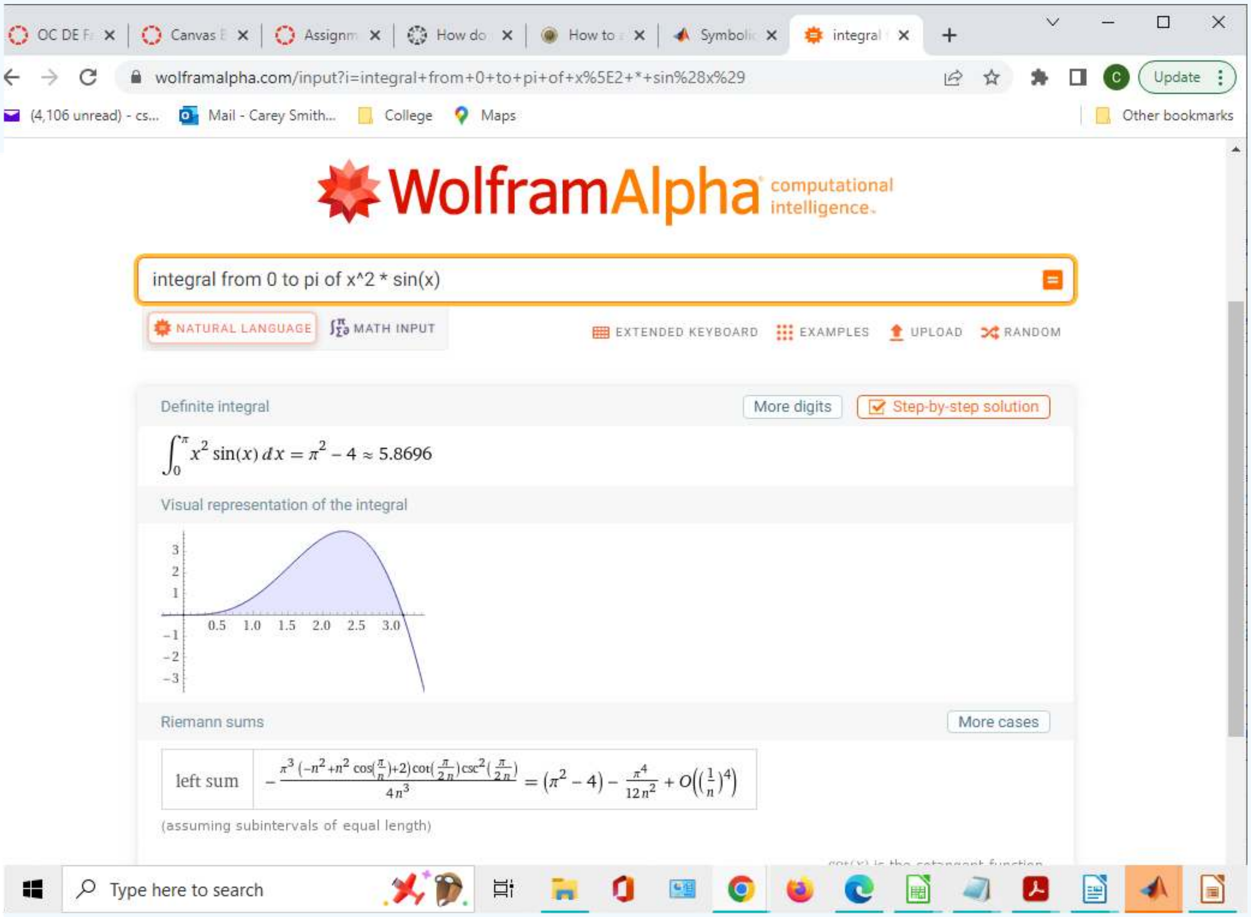
vpasolve
Numerical solution of a symbolic equation.


tave Tutorial #25: Solve Algebraic
STEM EDU TV
Tutorial
Symbolic Math
>> syms x y
Watch on YouTube

WolframAlpha

Symbolic processing can also be done for free at https://www.wolframalpha.com/

✓ Example 20.1 WolfranAlpha integral from 0 to pi of x^2 * sin(x)


OC DE F
Canvas
Assignm
How do
How to
Symbolic
integral
(4,106 unread) - cs...
Mail - Carey Smith...
College
Maps
Other bookmarks
Update
WolframAlpha computational intelligence.
NATURAL LANGUAGE
MATH INPUT
EXTENDED KEYBOARD
EXAMPLES
UPLOAD
RANDOM
More digits
Step-by-step solution
3
2
1
−1
−2
−3
0.5
1.0
1.5
2.0
2.5
3.0
More cases
Type here to search

wolframalpha.com/input?i=integral+from+0+to+pi+of+x%5E2+*+sin%28x%29

integral from 0 to pi of x^2 * sin(x)

Definite integral

$$\int_0^\pi x^2 \sin(x)\,dx = \pi^2 - 4 \approx 5.8696$$

Visual representation of the integral

Riemann sums

left sum	$-\frac{\pi^3\left(-n^2+n^2\cos\left(\frac{\pi}{n}\right)+2\right)\cot\left(\frac{\pi}{2n}\right)\csc^2\left(\frac{\pi}{2n}\right)}{4n^3} = \left(\pi^2-4\right) - \frac{\pi^4}{12n^2} + O\left(\left(\frac{1}{n}\right)^4\right)$

(assuming subintervals of equal length)

Solution

Add example text here.

This page titled 20: Symbolic Processing is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Carey Smith.

Detailed Licensing

Overview

Title: Matlab and Octave Programming for STEM Applications (Smith)

Webpages: 169

Applicable Restrictions: Noncommercial

All licenses found:

- CC BY-NC 4.0: 46.2% (78 pages)
- CC BY-NC-SA 4.0: 36.1% (61 pages)
- Undeclared: 12.4% (21 pages)
- CC BY-NC 3.0: 3% (5 pages)
- CC BY 4.0: 2.4% (4 pages)

By Page

- Matlab and Octave Programming for STEM Applications (Smith) — *CC BY-NC-SA 4.0*
    - Front Matter — *CC BY-NC-SA 4.0*
        - Authors — *Undeclared*
        - Info page — *Undeclared*
        - Title Page — *Undeclared*
    - 1: Modeling, Simulation, and MATLAB's Interpreter — *CC BY-NC 4.0*
        - 1.1: Modeling — *CC BY-NC 4.0*
        - 1.2: A Glorified Calculator — *CC BY-NC 4.0*
        - 1.3: Variables — *CC BY-NC 4.0*
        - 1.4: Errors — *CC BY-NC 4.0*
        - 1.5: Documentation — *CC BY-NC 4.0*
        - 1.6: Chapter Review — *CC BY-NC 4.0*
        - 1.7: Exercises — *CC BY-NC 4.0*
    - 2: MATLAB and OCTAVE User Interfaces and Scripts — *CC BY-NC 4.0*
        - 2.1: MATLAB User Interface — *CC BY-NC-SA 4.0*
        - 2.2: Octave User Interface — *CC BY-NC-SA 4.0*
        - 2.3: Writing Scripts to Solve Problems Beyenir — *CC BY 4.0*
        - 2.4: Your First Script — *CC BY-NC 4.0*
        - 2.5: Why Scripts? — *CC BY-NC 4.0*
        - 2.6: The Fibonacci Sequence — *CC BY-NC 4.0*
        - 2.7: Floating-Point Numbers — *CC BY-NC 4.0*
        - 2.8: Comments — *CC BY-NC 4.0*
        - 2.9: Documentation — *CC BY-NC 4.0*
        - 2.10: Assignment and Equality — *CC BY-NC 4.0*
        - 2.11: Complex Numbers — *CC BY-NC-SA 4.0*
        - 2.12: Chapter Review — *CC BY-NC 4.0*
        - 2.13: Exercises — *CC BY-NC 4.0*
    - 3: Loops — *CC BY-NC-SA 4.0*
        - 3.1: Updating Variables — *CC BY-NC 4.0*
        - 3.2: Bug Taxonomy — *CC BY-NC 4.0*
        - 3.3: Absolute and Relative Error — *CC BY-NC 4.0*
        - 3.4: for Loops — *CC BY-NC 4.0*
        - 3.5: for Loop Supplemental Material — *CC BY-NC-SA 4.0*
        - 3.6: Plotting — *CC BY-NC 4.0*
        - 3.7: Create a 2D x-y Sine-Wave Plot with a for Loop — *CC BY-NC-SA 4.0*
        - 3.8: Sequences — *CC BY-NC 4.0*
        - 3.9: Sequences Additional Material — *CC BY-NC-SA 4.0*
        - 3.10: Series Additional Material — *CC BY-NC-SA 4.0*
        - 3.11: Generalization — *CC BY-NC 4.0*
        - 3.12: Incremental Development — *CC BY-NC 4.0*
        - 3.13: for Loop Incremental Development Exercise — *CC BY-NC-SA 4.0*
        - 3.14: Chapter Review — *CC BY-NC 4.0*
        - 3.15: Exercises — *CC BY-NC 4.0*
    - 4: Vectors — *CC BY-NC-SA 4.0*
        - 4.1: Creating Vectors — *CC BY-NC 4.0*
        - 4.2: Methods to create vectors in MATLAB — *CC BY-NC-SA 4.0*
        - 4.3: Plotting Vectors Supplemental Material — *CC BY-NC-SA 4.0*
        - 4.4: Vector Arithmetic — *CC BY-NC 4.0*
        - 4.5: Selecting Elements — *CC BY-NC 4.0*
        - 4.6: Indexing Errors — *CC BY-NC 4.0*
        - 4.7: Vectors and Sequences — *CC BY-NC 4.0*
        - 4.8: disp() and num2str() Functions and Concatenation — *CC BY-NC-SA 4.0*
        - 4.9: Plotting Vectors — *CC BY-NC 4.0*
        - 4.10: Recursive for Loops — *CC BY-NC-SA 4.0*

    - 4.11: Common Vector Operations — *CC BY-NC 4.0*
    - 4.12: Common Vector Functions — *CC BY-NC-SA 4.0*
    - 4.13: MATLAB Commands and Functions — *CC BY-NC-SA 4.0*
    - 4.14: Chapter Review — *CC BY-NC 4.0*
    - 4.15: Exercises — *CC BY-NC 4.0*
    - 4.16: Working with Conditions — *CC BY-NC 4.0*
  - 5: User-Defined Functions — *Undeclared*
    - 5.1: Name Collisions — *CC BY-NC 4.0*
    - 5.2: Defining Functions — *CC BY-NC 4.0*
    - 5.3: Function Documentation — *CC BY-NC 4.0*
    - 5.4: Naming Functions — *CC BY-NC 4.0*
      - 5.4.1: Function Names and MATLAB Keywords — *Undeclared*
    - 5.5: Multiple Input Variables — *CC BY-NC 4.0*
      - 5.5.1: Multiple Inputs Supplemental Material — *Undeclared*
    - 5.6: Functions with Multiple Outputs — *Undeclared*
    - 5.7: Matlab Functions with No Inputs — *Undeclared*
    - 5.8: Matlab Functions with No Outputs — *Undeclared*
    - 5.9: Matlab Functions with No Inputs and No Outputs — *CC BY-NC-SA 4.0*
    - 5.10: Functions with variable numbers of inputs or outputs — *CC BY-NC-SA 4.0*
    - 5.11: Chapter Review — *CC BY-NC 4.0*
    - 5.12: Exercise — *CC BY-NC 4.0*
  - 6: Conditionals — *CC BY-NC 4.0*
    - 6.1: Relational Operators — *CC BY-NC 4.0*
    - 6.2: if Statement — *CC BY-NC 4.0*
    - 6.3: if, else, elseif, error() Parts 2 and 3 — *Undeclared*
    - 6.4: Incremental Development — *CC BY-NC 4.0*
    - 6.5: Logical Functions — *CC BY-NC 4.0*
    - 6.6: Nested Loops — *CC BY-NC 4.0*
    - 6.7: Putting It Together — *CC BY-NC 4.0*
    - 6.8: Encapsulation and Generalization — *CC BY-NC 4.0*
    - 6.9: Adding a continue Statement — *CC BY-NC 4.0*
    - 6.10: How Functions Work — *CC BY-NC 4.0*
    - 6.11: Chapter Review — *CC BY-NC 4.0*
    - 6.12: Exercise — *CC BY-NC 4.0*
  - 7: More Logical Functions, Series, Switch Logic, While Loops — *CC BY-NC-SA 4.0*
    - 7.1: if, elseif, else with Error Checking — *CC BY-NC-SA 4.0*
    - 7.2: Flowcharts — *Undeclared*
    - 7.3: Shortcut and, or, Logical Operations — *CC BY-NC-SA 4.0*
    - 7.4: The input Function — *CC BY-NC-SA 4.0*
    - 7.5: Compound Logical Operators — *CC BY-NC 4.0*
    - 7.6: if, else, elseif Part 2 — *Undeclared*
    - 7.7: for loops with break Logic — *CC BY-NC-SA 4.0*
    - 7.8: 'while' Loops — *CC BY-NC 3.0*
    - 7.9: 'switch' Logic — *CC BY-NC 3.0*
  - 8: User-defined Functions of Vectors — *CC BY-NC 4.0*
    - 8.1: Functions and Vectors — *CC BY-NC 4.0*
    - 8.2: Computing with Vectors — *CC BY-NC 4.0*
    - 8.3: Debugging in Four Acts — *CC BY-NC 4.0*
    - 8.4: Chapter Review — *CC BY-NC 4.0*
  - 9: MATLAB Random Number and Statistics Functions — *CC BY-NC-SA 4.0*
    - 9.1: MATLAB Random Number Functions — *CC BY-NC-SA 4.0*
    - 9.2: MATLAB Statistics Functions — *CC BY-NC-SA 4.0*
    - 9.3: Statistical Plots — *CC BY-NC-SA 4.0*
  - 10: Zero Finding and Fourier Transforms — *CC BY-NC-SA 4.0*
    - 10.1: Solving Nonlinear Equations — *CC BY-NC 4.0*
    - 10.2: Debugging — *CC BY-NC 4.0*
    - 10.3: Fourier Transforms — *Undeclared*
    - 10.4: Chapter Review — *CC BY-NC 4.0*
    - 10.5: Exercises — *CC BY-NC 4.0*
  - 11: MATLAB Matrices — *CC BY-NC-SA 4.0*
    - 11.1: Creating Matrices and Arrays — *CC BY-NC-SA 4.0*
    - 11.2: Matrices — *CC BY-NC 3.0*
    - 11.3: Using Matrices — *CC BY-NC 3.0*
    - 11.4: Matrix Operations — *CC BY-NC 3.0*
    - 11.5: Common Matrix Functions — *Undeclared*
    - 11.6: Matrix Tables and Nested Loops — *CC BY-NC-SA 4.0*
  - 12: Linear Algebra — *CC BY-NC-SA 4.0*
    - 12.1: Matrix Multiplication — *CC BY-NC-SA 4.0*
    - 12.2: Matrix Inverse — *CC BY-NC-SA 4.0*
    - 12.3: Systems of Equations — *CC BY-NC-SA 4.0*
    - 12.4: Systems of Equations Exercises part 1 — *CC BY-NC-SA 4.0*
    - 12.5: Systems of Equations Examples and Exercises, Part 2 — *CC BY-NC-SA 4.0*
    - 12.6: Over-Determined Systems of Equations — *CC BY-NC-SA 4.0*
  - 13: Formatted Input, Output — *CC BY-NC-SA 4.0*

    - 13.1: sprintf() and fprintf() Part 1 — *CC BY-NC-SA 4.0*
    - 13.2: fprintf() to a file — *CC BY 4.0*
    - 13.3: Reading from an Excel file in MATLAB and Octave — *Undeclared*
  - 14: Matlab Structures and Data Types — *CC BY-NC-SA 4.0*
    - 14.1: Table Data Type — *CC BY-NC-SA 4.0*
    - 14.2: struct Data Type — *Undeclared*
    - 14.3: struct Examples and Exercises — *CC BY-NC-SA 4.0*
    - 14.4: Numeric data types — *CC BY-NC-SA 4.0*
    - 14.5: Comparing Character Strings — *CC BY-NC-SA 4.0*
    - 14.6: cell array — *CC BY-NC-SA 4.0*
  - 15: Advanced Plotting — *CC BY-NC-SA 4.0*
    - 15.1: scatter, scatter3 — *Undeclared*
    - 15.2: Plot 3D Surface and Contour Plots — *CC BY-NC-SA 4.0*
    - 15.3: Advanced Plot Formatting — *Undeclared*
  - 16: Interpolation and Curve Fitting — *CC BY-NC-SA 4.0*
    - 16.1: Linear Interpolation — *CC BY-NC-SA 4.0*
      - 16.1.1: Interpolation Problem Set — *CC BY-NC-SA 4.0*
    - 16.2: Nonlinear Interpolation — *CC BY-NC-SA 4.0*
    - 16.3: Curve Fitting — *CC BY-NC-SA 4.0*
    - 16.4: Higher Order Polynomial Fitting — *CC BY-NC-SA 4.0*
    - 16.5: Regression Analysis — *CC BY-NC-SA 4.0*
      - 16.5.1: Problem Set — *CC BY-NC-SA 4.0*
    - 16.6: Interactive Fitting Tools — *CC BY-NC-SA 4.0*
  - 17: Object-Oriented Programming — *Undeclared*
    - 17.1: Top-Down Design, Agile Software Development, and Object-Oriented Programming in MATLAB — *CC BY-NC-SA 4.0*
  - 18: Integration and Differentiation — *Undeclared*
    - 18.1: Computing the Area Under a Curve — *CC BY 4.0*
    - 18.2: Problem Set — *CC BY 4.0*
  - 19: Ordinary Differential Equations — *CC BY-NC 4.0*
    - 19.1: Functions and Files — *CC BY-NC 4.0*
    - 19.2: Differential Equations — *CC BY-NC 4.0*
    - 19.3: Euler’s Method — *CC BY-NC 4.0*
    - 19.4: Implementing Euler’s Method — *CC BY-NC 4.0*
    - 19.5: Solving ODEs with ode45 — *CC BY-NC 4.0*
    - 19.6: Time Dependence — *CC BY-NC 4.0*
    - 19.7: What Could Go Wrong? — *CC BY-NC 4.0*
    - 19.8: Labeling Axes — *CC BY-NC 4.0*
    - 19.9: Chapter Review — *CC BY-NC 4.0*
    - 19.10: Exercise — *CC BY-NC 4.0*
  - Back Matter — *CC BY-NC-SA 4.0*
    - Index — *Undeclared*

MATLAB FUNCTIONS, COMMANDS AND SPECIAL CHARACTERS

A PDF summary ("Cheat Sheet") of Basic MATLAB functions can be downloaded from this website:

https://www.mathworks.com/content/dam/mathworks/fact-sheet/matlab-basic-functions-reference.pdf?s_v1=58562&elqem=4694181_EM_NA_DIR_25-02_MOE-CG-B

The following is also a summary of common MATLAB functions by Brian Vick, Virginia Tech. There are more functions for specific MATLAB add-on toolboxes, which are not listed here.

MATLAB has additional, less common functions. You can use MATLAB's help or Octave's documentation if you need a specialized function. Also "MATLAB Central" has user contributed functions. It is at:

https://www.mathworks.com/matlabcentral/

GENERAL PURPOSE COMMANDS

OPERATORS AND SPECIAL CHARACTERS

+ Plus; addition operator.
- Minus; subtraction operator.
* Scalar and matrix multiplication operator.
.* Array multiplication operator.
^ Scalar and matrix exponentiation operator.
.^ Array exponentiation operator.
\ Left-division matrix operator.
/ Right-division matrix operator.
.\ Array left-division operator.
./ Array right-division operator.
: Colon; generates regularly spaced elements and represents an entire row or column.
( ) Parentheses; encloses function arguments and array indices; overrides precedence.
[ ] Brackets; enclosures array elements.
. Decimal point.
… Ellipsis; line-continuation operator.
, Comma; separates statements and elements in a row.
; Semicolon; separates columns and suppresses display.
% Percent sign; designates a comment and specifies formatting.
' Quote sign and array or matrix transpose operator.
.' Nonconjugated array or matrix transpose operator.
= Assignment (replacement) operator.

COMMANDS FOR MANAGING A SESSION

clc: Clears Command window.
clear: Removes variables from memory.
exist: Checks for existence of file or variable.
global: Declares variables to be global.
help: Searches for a help topic.
lookfor: Searches help entries for a keyword.
quit: Stops MATLAB.
who: Lists current variables.
whos: Lists current variables (long display).

SPECIAL VARIABLES AND CONSTANTS

ans: Most recent answer.
eps: Accuracy of floating-point precision.
i,j: The imaginary unit -1.
Inf or inf: Infinity.

NaN or nan: Not a Number (Undefined numerical result).
pi: The irrational number 3.141592653589793...

SYSTEM AND FILE COMMANDS

cd: Changes current directory.
date: Displays current date.
delete: Deletes a file.
diary: Switches on/off diary file recording.
dir Lists all files in current directory.
load: Loads workspace variables from a file.
path: Displays search path.
pwd: Displays current directory.
save: Saves workspace variables in a file.
type: Displays contents of a file.
what: Lists all MATLAB files in the current directory.
wklread: Reads .wk1 spreadsheet file.

INPUT/OUTPUT AND FORMATTING COMMANDS

INPUT/OUTPUT COMMANDS

disp: Displays contents of an array or string.
fscanf: Read formatted data from a file.
format: Controls screen-display format.
fprintf: Performs formatted writes to screen or file.
input: Displays prompts and waits for input.
; Suppresses screen printing.

FORMAT CODES FOR FPRINTF AND FSCANF

%s Format as a string.
%d Format as an integer.
%f Format as a floating point value.
%e Format as a floating point value in scientific notation.
%g Format in the most compact form: %f or %e.
\n Insert a new line in the output string.
\t Insert a tab in the output string.

NUMERIC DISPLAY FORMATS

format short	Four decimal digits (default).
format long	16 decimal digits.
format short e	Five digits plus exponent.
format long e	16 digits plus exponents.
format bank	Two decimal digits.
format +	Positive, negative, or zero.
format rat	Rational approximation.
format compact	Suppresses some line feeds.
format loose	Resets to less compact display mode.

VECTOR, MATRIX AND ARRAY COMMANDS

ARRAY COMMANDS

cat: Concatenates arrays.
find: Finds indices of nonzero elements.
length: Computers number of elements.
linspace: Creates regularly spaced vector.
logspace: Creates logarithmically spaced vector.

max: Returns largest element.
min: Returns smallest element.
prod: Product of each column.
reshape: Change size
size: Computes array size.
sort: Sorts each column.
sum: Sums each column.

SPECIAL MATRICES

eye: Creates an identity matrix.
ones: Creates an array of ones.
zeros: Creates an array of zeros.

MATRIX ARITHMETIC

cross: Computes cross products.
dot: Computes dot products.

MATRIX COMMANDS FOR SOLVING LINEAR EQUATIONS

det: Computes determinant of an array.
inv: Computes inverse of a matrix.
pinv: Computes pseudoinverse of a matrix.
rank: Computes rank of a matrix.
rref: Computes reduced row echelon form.

CELL ARRAY FUNCTIONS

cell: Creates cell array.
celldisp: Displays cell array.
cellplot: Displays graphical representation of cell array.
num2cell: Converts numeric array to cell array.
deal: Matches input and output lists.
iscell: Identifies cell array.

STRUCTURE FUNCTIONS

fieldnames: Returns field names in a structure array.
getfield: Returns field contents of a structure array.
isfield: Identifies a structure array field.
isstruct: Identifies a structure array.
rmfield: Removes a field from a structure array.
setfield: Sets contents of field.
struct: Creates structure array.

PLOTTING COMMANDS

BASIC XY PLOTTING COMMANDS

axis: Sets axis limits.
fplot: Intelligent plotting of functions.
grid: Displays gridlines.
plot: Generates xy plot.
print: Prints plot or saves plot to a file
title: Puts text at top of plot.
xlabel: Adds text label to x-axis.
ylabel: Adds text label to y-axis.

PLOT ENHANCEMENT COMMANDS

axes: Creates axes objects.
close: Closes the current plot.

close all: Closes all plots.
figure: Opens a new figure window.
gtext: Enables label placement by mouse.
hold: Freezes current plot.
legend: Create a plot legend
refresh: Redraws current figure window.
set: Specifies properties of objects such as axes.
subplot: Creates plots in subwindows.
text: Places string in figure.

SPECIALIZED PLOT COMMANDS

bar: Creates bar chart.
loglog: Creates log-log plot.
polar: Creates polar plot.
semilogx: Creates semilog plot (logarithmic abscissa).
semilogy: Creates semilog plot (logarithmic ordinate).
stairs: Creates stairs pot.
stem: Creates stem plot.

COLORS, SYMBOLS AND LINE TYPES

COLOR

k black

b blue

c cyan

m magenta

g green

r red

w white

y yellow

SYMBOL

. point

o circle

x x-mark

+ plus

* star

d diamond

v triangle (down)

^ triangle (up)

< triangle (left)

> triangle (right)

p pentagram

h hexagram

LINE

- solid

: dotted

-. dash dotted

-- dashed

THREE-DIMENSIONAL PLOTTING COMMANDS

contour: Creates contour plot.
mesh: Creates three-dimensional mesh surface plot.
meshc: Same as mesh with contour plot underneath.
meshz: Same as mesh with vertical lines underneath.
plot3: Creates three-dimensional plots from lines and points.
surf: Creates shaded three-dimensional mesh surface plot.
surfc: Same as surf with contour plot underneath.
meshgrid: Creates rectangular grids of x and y values.
waterfall: Same as mesh with mesh lines in one direction.
zlabel: Adds text label to z-axis.

HISTOGRAM FUNCTIONS

bar: Creates a bar chart.
hist: Aggregates the data into equally spaced bins. (old version)
histogram: Aggregates the data into equally spaced bins. (new version)
histc: Aggregates the data into unequally spaced bins.

PROGRAMMING

LOGICAL AND RELATIONAL OPERATORS

== Relational operator: equal to.
~= Relational operator: not equal to.
< Relational operator: less than.
<= Relational operator: less than or equal to.
> Relational operator: greater than.
>= Relational operator: greater than or equal to.
& Logical operator: AND.
| Logical operator: OR.
~ Logical operator: NOT.
xor Logical operator: EXCLUSIVE OR.

PROGRAM FLOW CONTROL

break: Terminates execution of a loop.
case: Provides alternate execution paths within switch structure.
else: Delineates alternate block of statements.
elseif: Conditionally executes statements.
end: Terminates for, while, and if statements.
error: Display error messages.
for: Repeats statements a specific number of times
if: Executes statements conditionally.
otherwise: Default part of switch statement.
return: Return to the invoking function.
switch: Directs program execution by comparing point with case expressions.
warning: Display a warning message.
while: Repeats statements an indefinite number of times.

LOGICAL FUNCTIONS

any: True if any elements are nonzero.
all: True if all elements are nonzero.
find: Finds indices of nonzero elements.
finite: True if elements are finite.
isnan: True if elements are undefined.
isinf: True if elements are infinite.

isempty: True if matrix is empty.
isreal: True if all elements are real.

M-FILES

eval: Interpret strings containing Matlab expressions.
feval: Function evaluation.
function: Creates a user-defined function M-file.
global: Define global variables.
nargin: Number of function input arguments.
nargout: Number of function output arguments.
script: Script M-files

TIMING

cputime: CPU time in seconds.
clock: Current date and time as date vector.
tic, toc: Start, stop a stopwatch timer.

MATHEMATICAL FUNCTIONS

EXPONENTIAL AND LOGARITHMIC FUNCTIONS

exp(x): Exponential; ex.
log(x): Natural logarithm; ln(x).
log10(x): Common (base 10) logarithm;

TRIGONOMETRIC FUNCTIONS

acos(x) Inverse cosine; arcos x = cos –1 (x).
acot(x) Inverse cotangent; arccot x = cot –1(x).
acsc(x) Inverse cosecant; arcs x = csc –1 (x).
asec(x) Inverse secant; arcsec x = sec –1 (x).
asin(x) Inverse sine; arcsin x = sin –1 (x).
atan(x) Inverse tangent; arctan x = tan –1 (x).
atan2(y,x) Four-quadrant inverse tangent.
cos(x) Cosine; cos(x).
cot(x) Cotangent; cot(x).
csc(x) Cosecant; csc(x).
sec(x) Secant; sec(x).
sin(x) Sine; sin(x).
tan(x) Tangent; tan(x).

HYPERBOLIC FUNCTIONS

acosh(x) Inverse hyperbolic cosine; cosh –1 (x).
acoth(x) Inverse hyperbolic cotangent; coth –1 (x).
acsch(x) Inverse hyperbolic cosecant; csch –1 (x).
asech(x) Inverse hyperbolic secant; sech –1 (x).
asinh(x) Inverse hyperbolic sine; sinh –1 (x).
atanh(x) Inverse hyperbolic tangent; tanh –1 (x).
cosh(x) Hyperbolic cosine; cosh(x).
coth(x) Hyperbolic cotangent; cosh(x)/sinh(x).
csch(x) Hyperbolic cosecant; 1/sinh(x).
sech(x) Hyperbolic secant; 1/cosh(x).
sinh(x) Hyperbolic sine; sinh(x).
tanh(x) Hyperbolic tangent; sinh(x)/cosh(x).

COMPLEX FUNCTIONS

abs(x) Absolute value; |x|.
angle(x) Angle of a complex number x.
conj(x) Complex conjugate of x.
imag(x) Imaginary part of a complex number x.
real(x) Real part of a complex number x.

STATISTICAL FUNCTIONS

erf(x): Computes the error function erf (x).
mean: Calculates the average.
median: Calculates the median.
std: Calculates the standard deviation.

RANDOM NUMBER FUNCTIONS

rand: Generates uniformly distributed random numbers between 0 and 1.
randi(IMAX): Generates uniformly distributed random integers between 1 and IMAX.
randn: Generates normally distributed random numbers.

NUMERIC FUNCTIONS

ceil: Rounds to the nearest integer toward +infinity.
fix: Rounds to the nearest integer toward zero.
floor: Rounds to the nearest integer toward infinity.
round: Rounds towards the nearest integer.
sign: Signum function. (-1, 0, or 1)

STRING FUNCTIONS

findstr: Finds occurrences of a string.
strcmp: Compares strings.
char: Creates character string array

NUMERICAL METHODS

POLYNOMIAL AND REGRESSION FUNCTIONS

conv: Computes product of two polynomials (convolution)
deconv: Computes ratio of polynomials.
eig: Computes the eigenvalues of a matrix.
poly: Computes polynomial from roots.
polyfit: Fits a polynomial to data.
polyval: Evaluates polynomial and generates error estimates.
roots: Computes polynomial roots.

INTERPOLATION FUNCTIONS

interp1: Linear and cubic-spline interpolations of a function of one variable.
interp2: Linear interpolation of a function of two variables.
spline: Cubic-spline interpolation.
unmkpp: Computes the coefficients of cubic-spine polynomials.

ROOT FINDING AND MINIMIZATION

fmin: Finds minimum of single-variable function.
fmins: Finds minimum of multivariable function.
fzero: Finds zero of single-variable function.

NUMERICAL INTEGRATION FUNCTIONS

integral: Numerical integration (newer, nominal default)

quad: Numerical integration with adaptive Simpson’s rule. (older)
quadl: Numerical integration with adaptive Lobatto quadrature. (older)

trapz: Numerical integration with the trapezoidal rule.

NUMERICAL DIFFERENTIATION FUNCTIONS

diff(x): Computes the difference between adjacent elements in the vector x.
polyder: Differentiates a polynomial, a polynomial product, or a polynomial quotient.

gradient(y,x): Computes dy/dx

ODE SOLVERS

ode23: Nonstiff, low-order solver.
ode45: Nonstiff, medium-order solver. (most common)
ode113: Nonstiff, variable-order solver.
ode23s: Stiff, low-order.
ode23t: Moderately stiff, trapezoidal rule solver.
ode23b: Stiff, low-order solver.
ode15s: Stiff, variable-order solver.
odeset: Creates integrator options structure for ODE solvers.

PREDEFINED INPUT FUNCTIONS

gensig: Generates a periodic sine, square, or pulse input.
sawtooth: Generates a periodic sawtooth input.
square: Generates a square wave input.
stepfun: Generates a step function input.

Index

A

- Absolute error
  - 3.3: Absolute and Relative Error
  - 3.11: Chapter Review
- accumulator
  - 3.11: Chapter Review
- add1
  - 5.5.1: Multiple Inputs Examples and Exercises
- add2
  - 6.3: if, else, elseif
- Agile Software Development
  - 17.1: Top-Down Design
- Airy pattern
  - 3.7.1: Series Additional Material
- and operator
  - 7.4: Compound Logical Operators
  - 7.4.1: Compound if, elseif Logic Examples
  - 7.5: Working with Conditions
  - 7.6: Shortcut and, or, Logical Operations
- anonymous function
  - 11.5: Anonymous (or In-Line) Functions
  - 19.5.1: ODE45 Examples and Exercises
- Apple
  - 1.1: Obtaining MATLAB and Octave
- area
  - 2.3: Examples of assignment scripts
- Area of a Square
  - 6.3: if, else, elseif
- arithmetic
  - 1.11: Chapter 1 videos English and Espanol
- arithmetic operators
  - 1.3: A Glorified Calculator
- Array
  - 12.1: Creating Matrices and Arrays
- array index
  - 4.5: Indexing Errors
- ASCII
  - 4.7: disp() and num2str() Functions and Concatenation
- assignment
  - 2.12: Assignment and Equality
  - 5.2.1: Function Case Study (Lambert)
- asymptotic
  - 3.8: Generalization
- atan
  - 1.3.2: Trigonometry Functions
  - 3.7.1: Series Additional Material
- atan2
  - 1.3.2: Trigonometry Functions
  - 5.5.1: Multiple Inputs Examples and Exercises
- axes
  - 19.8: Labeling Axes
- axis
  - 14.1: scatter, scatter3
- axis equal
  - 14.1: scatter, scatter3

B

- ball
  - 4.9: Recursive for Loops
- Basketball team
  - 16.4.1: struct Examples and Exercises
- beep
  - 7.4.1: Compound if, elseif Logic Examples
- Bessel Function
  - 3.7.1: Series Additional Material
- bike
  - 2.17: Exercises
  - 3.1: Updating Variables
  - 3.4: for Loops
- Bike for loop
  - 3.4.1: for Loop Supplemental Material
- Bike_update
  - 3.4.1: for Loop Supplemental Material
- bikeloop
  - 3.5: Plotting Points
- body of a loop
  - 3.11: Chapter Review
- Body temperature
  - 7.4.1: Compound if, elseif Logic Examples
- boxplot
  - 9.3: Statistical Plots
- break
  - 7.5: Working with Conditions
  - 7.7: for loops with break Logic
  - 7.8: 'while' Loops
- Bug Taxonomy
  - 3.2: Bug Taxonomy
- bugs
  - 3.2: Bug Taxonomy

C

- cadet
  - 7.9: 'switch' Logic
- calculator
  - 1.3: A Glorified Calculator
- Cambridge
  - 3.1: Updating Variables
- Camrosa
  - 9.2: MATLAB Statistics Functions
- case
  - 7.9: 'switch' Logic
- catch
  - 7.1: if, elseif, else with Error Checking
- ceil
  - 1.3.1: log(), round(), floor(), and ceil() functions
- cell array
  - 16.5: cell array
- character string
  - 15.1: sprintf() and fprintf(), Introduction
  - 16.5: cell array
- Chromebook
  - 1.1: Obtaining MATLAB and Octave
- circuit
  - 13.5: Systems of Equations Examples and Exercises
- class
  - 17.4: Object-Oriented Programming, Part 2
- clc
  - 1.7: Clearing the Workspace, Readability Tips
  - 5.9: Matlab Functions with No Inputs and No Outputs
- clear
  - 1.7: Clearing the Workspace, Readability Tips
  - 4.11: Common Vector Functions
- Clear all
  - 2.14: Debugging Directory Errors
- clf
  - 3.5: Plotting Points
- close
  - 1.7: Clearing the Workspace, Readability Tips
- CO2
  - 4.11: Common Vector Functions
- cobblestone
  - 2.17.1: More Exercises
- colon operator
  - 3.4: for Loops
  - 4.1: Creating Vectors
  - 4.2: Methods to Create Vectors in MATLAB
- colorbar
  - 5.9: Matlab Functions with No Inputs and No Outputs
- colormap
  - 14.2: Plot 3D Surface and Contour Plots
- column
  - 4.11: Common Vector Functions
- commands
  - MATLAB Functions, Commands and Special Characters
- Commas
  - 2.7: Scientific Notation for Large and Small Numbers
  - 4.8.1: Plotting Vectors Additional Material
- comment
  - 2.3: Examples of assignment scripts
  - 2.10: Script Comments
- comments
  - 2.11: Script Documentation
- commutative
  - 13.3: Matrix Inverse, Rank and Determinant
- complex
  - 2.9: Complex Numbers
- Compound if, elseif
  - 7.4.1: Compound if, elseif Logic Examples
- Compound Logical Operators
  - 7.4: Compound Logical Operators
- concatenation
  - 4.2: Methods to Create Vectors in MATLAB
  - 4.7: disp() and num2str() Functions and Concatenation
  - 4.9: Recursive for Loops
  - 7.3: The input() Function
- conditional
  - 7.10: The find() function
- conditional statement
  - 6.3: if, else, elseif
  - 7.2: Flowcharts
- conj
  - 2.9: Complex Numbers
  - 11.7: Fourier Transforms
- consistent
  - 13.6: Over-Determined Systems of Equations
- Console
  - 5.9: Matlab Functions with No Inputs and No Outputs
- continue
  - 6.9: Adding a continue Statement
  - 7.8: 'while' Loops

contour
  14: Advanced Plotting
  14.2: Plot 3D Surface and Contour Plots
converge
  7.7: for loops with break Logic
cos
  1.3.2: Trigonometry Functions
  4.11: Common Vector Functions
cosd
  1.3.2: Trigonometry Functions
  4.11: Common Vector Functions
Countries
  16.5: cell array
cross product
  12.3: Common Matrix Functions
  12.8: Cross Product and Torque
current
  10.3: Linear and Polynomial Curve Data Fitting
Curve Fitting
  10.3: Linear and Polynomial Curve Data Fitting
  10.4: Polynomial Fitting Problem Set

D
Damped spring
  19.5.1: ODE45 Examples and Exercises
date
  5.7: Matlab Functions with No Inputs
Debug
  2.13: Understanding Editor Warning and Error messages
  2.14: Debugging Directory Errors
debuggable
  1.3: A Glorified Calculator
debugging
  8.3.1: Debugging Theorems all in 1 Place
Debugging 1st theorem of Debugging
  1.3: A Glorified Calculator
Debugging 2nd Theorem of Debugging
  1.8: Errors
Debugging 3rd Theorem of Debugging
  2.4: Why Scripts?
Debugging 4th Theorem of Debugging
  2.5: The Fibonacci Sequence
Debugging 6th Theorem of Debugging
  3.9: Incremental Development
Debugging 7th Theorem of Debugging
  11.6: Debugging- More Name Collisions
decimal places
  15.1: sprintf() and fprintf(), Introduction
derivative
  18.2: Numerical Differentiation
det
  13.3: Matrix Inverse, Rank and Determinant
determinant
  12.3: Common Matrix Functions
  13.3: Matrix Inverse, Rank and Determinant
diag
  12.1: Creating Matrices and Arrays
dice
  9.3: Statistical Plots
  16.5: cell array
differences
  8.1: Functions and Vectors
differential equation
  19.2: Differential Equations
Directory
  2.14: Debugging Directory Errors
disp()
  4.7: disp() and num2str() Functions and Concatenation
  7.3: The input() Function
diverge
  7.7: for loops with break Logic
divergent sequence
  7.4.1: Compound if, elseif Logic Examples
documentation
  2.11: Script Documentation
dot product
  12.3: Common Matrix Functions
  12.7: Dot Product
double precision
  16.1: Numeric data types
Douglas
  4.7: disp() and num2str() Functions and Concatenation

E
earthquake
  16.4.1: struct Examples and Exercises
Editor error
  2.13: Understanding Editor Warning and Error messages
element
  4.4: Selecting Elements
else
  6.3: if, else, elseif
  6.5: Logical Functions
  7.1: if, elseif, else with Error Checking
  7.2: Flowcharts
  7.9: 'switch' Logic
elseif
  6.3: if, else, elseif
  7.1: if, elseif, else with Error Checking
  7.2: Flowcharts
  7.9: 'switch' Logic
encapsulation
  6.8: Encapsulation and Generalization
end
  3.4: for Loops
  3.4.1: for Loop Supplemental Material
  6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction
endfunction
  6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction
endif
  6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction
engine cylinder
  18.1.1: Problem Set
equal
  6.1.1: Relational Operators List
  7.6: Shortcut and, or, Logical Operations
equality
  2.12: Assignment and Equality
equality operator
  6.1: Relational Operators
error
  4.5: Indexing Errors
  6.3: if, else, elseif
  7.1: if, elseif, else with Error Checking
errors
  1.8: Errors
  2.14: Debugging Directory Errors
Excel
  15.3: Reading from an Excel file in MATLAB and Octave
Exercise 2.2 Bike Swap
  2.17: Exercises
exist
  1.6.1: More About Valid Variable Names
Exponent Notation
  2.7: Scientific Notation for Large and Small Numbers
Exponetial Curve Fitting
  10.3: Linear and Polynomial Curve Data Fitting
extrapolation
  10.3: Linear and Polynomial Curve Data Fitting
eye
  12.1: Creating Matrices and Arrays
  12.3: Common Matrix Functions

F
factorial
  3.4.1: for Loop Supplemental Material
factorial function
  3.7.1: Series Additional Material
False
  6.3: if, else, elseif
fclose
  15.2: fprintf() to a file
FFT
  11: Zero Finding and Fourier Transforms
  11.7: Fourier Transforms
Fibonacci
  2.5: The Fibonacci Sequence
  3.6.1: Sequences Additional Material
  4.8: Plotting Vectors
  4.14: Exercises
Fibonacci sequence
  4.9: Recursive for Loops
figure
  3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop
  5.8: Matlab Functions with No Outputs
filename
  5.4.1: Function Names and MATLAB Keywords
filter
  9.4: Random Number Applications
find
  7.10: The find() function
Fitting by Transforming the Data
  10.3: Linear and Polynomial Curve Data Fitting
floor
  1.3.1: log(), round(), floor(), and ceil() functions
flowchart
  3.4.1: for Loop Supplemental Material
  7.2: Flowcharts
Folder
  2.14: Debugging Directory Errors
font
  1.4: MATLAB User Interface
  1.5: Octave User Interface
fopen
  15.2: fprintf() to a file

for loop
- 3.4: for Loops
- 3.4.1: for Loop Supplemental Material
- 3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop
- 3.6: Sequences
- 3.6.1: Sequences Additional Material
- 3.10: for Loop Incremental Development Exercise
- 4.6: Vectors and Sequences
- 4.9: Recursive for Loops
- 7.5: Working with Conditions
- 7.7: for loops with break Logic

format
- 2.8: Number formats
- 15.1: sprintf() and fprintf(), Introduction

format compact
- 1.7: Clearing the Workspace, Readability Tips

Fourier transform
- 11: Zero Finding and Fourier Transforms
- 11.7: Fourier Transforms

fprintf
- 15.1: sprintf() and fprintf(), Introduction
- 15.2: fprintf() to a file

Frederick
- 4.7: disp() and num2str() Functions and Concatenation

friction
- 4.9: Recursive for Loops

Fuel economy
- 10.3: Linear and Polynomial Curve Data Fitting

function
- 5.2: Defining Functions
- 5.3: Function Documentation
- 5.4.1: Function Names and MATLAB Keywords
- 5.6: Functions with Multiple Outputs
- 5.10: Functions with variable numbers of inputs or outputs

function declaration
- 5.2.1: Function Case Study (Lambert)

function inputs
- 5.3: Function Documentation

function outputs
- 5.3: Function Documentation

function signature
- 5.2: Defining Functions

functions
- 5.8: Matlab Functions with No Outputs
- 8.1.2: User-Defined Functions of Vectors Examples and Exercises
- 19.1: Functions and Files
- MATLAB Functions, Commands and Special Characters

Functions Summary
- MATLAB Functions, Commands and Special Characters

fzero
- 11: Zero Finding and Fourier Transforms
- 11.1: Solving Nonlinear Equations with fzero()
- 11.2: fzero() Examples and Exercises
- 11.3: Using fzero() to find the intersection of 2 functions

G

Gamma function
- 3.7.1: Series Additional Material

Generalization
- 3.11: Chapter Review
- 6.8: Encapsulation and Generalization

geometric sequence
- 4.14: Exercises

grade
- 6.3: if, else, elseif

gradient
- 18.2: Numerical Differentiation

Graphical Solution
- 13.5: Systems of Equations Examples and Exercises

graphics_toolkit
- 3.5: Plotting Points
- 9.3: Statistical Plots

graphics_toolkit("fltk")
- 3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop

greater than
- 6.1.1: Relational Operators List
- 7.6: Shortcut and, or, Logical Operations

greater than operator
- 6.1: Relational Operators

greater than or equal
- 7.6: Shortcut and, or, Logical Operations

greater than or equal to
- 6.1.1: Relational Operators List

growth rate
- 19.6: Time Dependence

H

helix
- 14.1: scatter, scatter3

help
- 4.9: Recursive for Loops

helper function
- 19.1: Functions and Files

hist
- 9.3: Statistical Plots

histogram
- 9.3: Statistical Plots

hold
- 3.5: Plotting Points

hyphen
- 5.4.1: Function Names and MATLAB Keywords

I

i
- 2.9: Complex Numbers

ideal gas law
- 5.5.2: Mutiple Inputs Example- Ideal Gas law (Lambert)

if
- 6.3: if, else, elseif
- 6.5: Logical Functions
- 7.1: if, elseif, else with Error Checking
- 7.2: Flowcharts
- 7.5: Working with Conditions
- 7.7: for loops with break Logic
- 7.9: 'switch' Logic

imag
- 2.9: Complex Numbers

inconsistent
- 13.6: Over-Determined Systems of Equations

increment
- 4.2: Methods to Create Vectors in MATLAB

Incremental development
- 3.9: Incremental Development
- 3.10: for Loop Incremental Development Exercise
- 3.11: Chapter Review

index
- 3.4.1: for Loop Supplemental Material
- 5.6: Functions with Multiple Outputs

indexing
- 4.5: Indexing Errors

indices
- 7.10: The find() function

input()
- 7.3: The input() Function

inputs
- 5.5.1: Multiple Inputs Examples and Exercises
- 5.7: Matlab Functions with No Inputs
- 5.8: Matlab Functions with No Outputs
- 5.9: Matlab Functions with No Inputs and No Outputs
- 5.10: Functions with variable numbers of inputs or outputs

integer
- 16.1: Numeric data types

integral function
- 18.1: Integration- Computing the Area Under a Curve

integral()
- 18.1.2: The integral() function

integration
- 18.1: Integration- Computing the Area Under a Curve

interp1
- 10.1: Linear Interpolation
- 10.2: Nonlinear Interpolation

interp2
- 10.2: Nonlinear Interpolation

interpolation
- 10.1: Linear Interpolation
- 10.1.1: Linear Interpolation Problem Set
- 10.2: Nonlinear Interpolation

interpreter
- 1.3: A Glorified Calculator

intersection
- 11.3: Using fzero() to find the intersection of 2 functions

inverse
- 12.3: Common Matrix Functions

iskeyword
- 1.6.1: More About Valid Variable Names

isosceles
- 2.3: Examples of assignment scripts

iterations
- 4.9: Recursive for Loops

J

j
- 2.9: Complex Numbers

K

keyword
- 5.4.1: Function Names and MATLAB Keywords

Kirchhoff's Voltage Law
- 13.5: Systems of Equations Examples and Exercises

L

Large Numbers
- 2.7: Scientific Notation for Large and Small Numbers

legend
  4.9: Recursive for Loops
  10.3: Linear and Polynomial Curve Data Fitting
  19.8: Labeling Axes
length
  2.9: Complex Numbers
  4.4: Selecting Elements
  4.11: Common Vector Functions
  7.5: Working with Conditions
  12.3: Common Matrix Functions
less than
  6.1.1: Relational Operators List
  7.6: Shortcut and, or, Logical Operations
less than operator
  6.1: Relational Operators
less than or equal
  7.6: Shortcut and, or, Logical Operations
less than or equal to
  6.1.1: Relational Operators List
Life Cycle Matrix
  13.1: Basic Matrix Multiplication
  13.2: Matrix Multiplication Applications
Linear Fit
  10.3: Linear and Polynomial Curve Data Fitting
LineWidth
  4.9: Recursive for Loops
linspace
  4.2: Methods to Create Vectors in MATLAB
log
  1.3.1: log(), round(), floor(), and ceil() functions
  4.11: Common Vector Functions
log10
  1.3.1: log(), round(), floor(), and ceil() functions
  4.11: Common Vector Functions
log2
  1.3.1: log(), round(), floor(), and ceil() functions
logical
  6.5: Logical Functions
Logical error
  3.2: Bug Taxonomy
logical operators
  6.3: if, else, elseif
  7.4: Compound Logical Operators
  7.6: Shortcut and, or, Logical Operations
logistic map
  4.14: Exercises
loglog
  14.4: Log plots
logspace
  4.2: Methods to Create Vectors in MATLAB
loop
  3.11: Chapter Review
loop variable
  3.11: Chapter Review
Lorenz attractor
  4.14: Exercises

M
magic
  12.1: Creating Matrices and Arrays
magnitude
  12.7: Dot Product
major
  7.9: 'switch' Logic
MarkerFaceColor
  14.1: scatter, scatter3
MarkerSize
  14.1: scatter, scatter3
math functions
  1.3: A Glorified Calculator
matlab
  1.1: Obtaining MATLAB and Octave
MATLAB & Simulink Student Suite
  1.1: Obtaining MATLAB and Octave
MATLAB Development Environment
  1.11: Chapter 1 videos English and Espanol
matrix
  7.5: Working with Conditions
  12.1: Creating Matrices and Arrays
  12.4: Matrix Indexing
Matrix inverse
  13.3: Matrix Inverse, Rank and Determinant
  13.5: Systems of Equations Examples and Exercises
Matrix multiplication
  13.1: Basic Matrix Multiplication
matrix operations
  12.5: Matrix Operations, Introduction
Matrix Powers
  13.1: Basic Matrix Multiplication
  13.2: Matrix Multiplication Applications
matrix table
  12.6: Matrix Tables and Nested Loops
max
  4.11: Common Vector Functions
  5.6: Functions with Multiple Outputs
  5.10: Functions with variable numbers of inputs or outputs
  9.2: MATLAB Statistics Functions
  12.3: Common Matrix Functions
mean
  4.11: Common Vector Functions
  5.6: Functions with Multiple Outputs
  9.2: MATLAB Statistics Functions
Measurement errors
  10.3: Linear and Polynomial Curve Data Fitting
median
  4.11: Common Vector Functions
  9.2: MATLAB Statistics Functions
menu
  7.9: 'switch' Logic
mesh
  14: Advanced Plotting
  14.2: Plot 3D Surface and Contour Plots
meshgrid
  14: Advanced Plotting
  14.2: Plot 3D Surface and Contour Plots
message
  15.1: sprintf() and fprintf(), Introduction
min
  4.11: Common Vector Functions
  9.2: MATLAB Statistics Functions
  12.3: Common Matrix Functions
mlx
  2.2.1: .mlx Live Editor Files
multiple function inputs
  5.5.1: Multiple Inputs Examples and Exercises
multiple inputs
  5.5.2: Mutiple Inputs Example- Ideal Gas law (Lambert)
my_min
  8.1.2: User-Defined Functions of Vectors Examples and Exercises
mydate
  17.4: Object-Oriented Programming, Part 2

N
name
  5.4.1: Function Names and MATLAB Keywords
name collision
  5.1: Name Collisions
  5.4.1: Function Names and MATLAB Keywords
  11.6: Debugging- More Name Collisions
nested loops
  6.6: Nested for Loops
  12.6: Matrix Tables and Nested Loops
nonlinear equations
  11.1: Solving Nonlinear Equations with fzero()
nonlinear interpolation
  10.2: Nonlinear Interpolation
norm
  12.7: Dot Product
not equal
  6.1.1: Relational Operators List
not operator
  7.4: Compound Logical Operators
not ~
  7.6: Shortcut and, or, Logical Operations
num2str
  4.9: Recursive for Loops
num2str()
  4.7: disp() and num2str() Functions and Concatenation
  7.3: The input() Function
numel
  4.11: Common Vector Functions
numeric data types
  16.1: Numeric data types
numerical differentiation
  18.2: Numerical Differentiation
Numerical error
  3.2: Bug Taxonomy
numerical errors
  3.3: Absolute and Relative Error
numerical integration
  18.1.1: Problem Set
Nyquist–Shannon
  11.7: Fourier Transforms

O
object
  17.4: Object-Oriented Programming, Part 2
Octave
  1.1: Obtaining MATLAB and Octave
  1.3.1: log(), round(), floor(), and ceil() functions
  1.5: Octave User Interface
  3.5: Plotting Points
  6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction
Octave plot
  3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop
ODE
  19.2: Differential Equations
  19.5: Solving ODEs with ode45
ode45
  19.5: Solving ODEs with ode45
  19.5.1: ODE45 Examples and Exercises

ones
4.2: Methods to Create Vectors in MATLAB
12.1: Creating Matrices and Arrays
or operator
7.4: Compound Logical Operators
7.4.1: Compound if, elseif Logic Examples
7.6: Shortcut and, or, Logical Operations
ordinary differential equation
19.2: Differential Equations
otherwise
7.9: 'switch' Logic
outputs
5.6: Functions with Multiple Outputs
5.7: Matlab Functions with No Inputs
5.8: Matlab Functions with No Outputs
5.9: Matlab Functions with No Inputs and No Outputs
5.10: Functions with variable numbers of inputs or outputs
overdetermined system of equations
13.6: Over-Determined Systems of Equations

P
Parabola
4.9: Recursive for Loops
19.5.1: ODE45 Examples and Exercises
parallel
12.8: Cross Product and Torque
parentheses
5.4.1: Function Names and MATLAB Keywords
pascal
12.1: Creating Matrices and Arrays
pchip
10.2: Nonlinear Interpolation
PDE
19.2: Differential Equations
peaks
18.2: Numerical Differentiation
perpendicular
12.8: Cross Product and Torque
pi
5.7: Matlab Functions with No Inputs
plot
3.5: Plotting Points
3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop
4.8.1: Plotting Vectors Additional Material
12.7: Dot Product
plot color
3.5: Plotting Points
plot points
3.5: Plotting Points
plot style string
3.5: Plotting Points
plot vector
4.8: Plotting Vectors
plot3
12.7: Dot Product
12.8: Cross Product and Torque
polyfit
10.3: Linear and Polynomial Curve Data Fitting
polynomial fitting
10.4: Polynomial Fitting Problem Set
Polynoomial Curve Fitting
10.3: Linear and Polynomial Curve Data Fitting
polyval
10.3: Linear and Polynomial Curve Data Fitting
Population Growth
4.9: Recursive for Loops
postcondition
2.15: Chapter Review
5.3: Function Documentation
power peries
3.7.1: Series Additional Material
precondition
2.15: Chapter Review
5.3: Function Documentation
prod
12.3: Common Matrix Functions
pwd
2.14: Debugging Directory Errors
pythagorean
6.5: Logical Functions
Pythagorean triples
6.7: Putting It Together

Q
quiver
14.1: scatter, scatter3
18.2: Numerical Differentiation

R
rain
7.2: Flowcharts
rand
9.1: MATLAB Random Number Functions
14.1: scatter, scatter3
randi
9.1: MATLAB Random Number Functions
randn
9.1: MATLAB Random Number Functions
random
9.4: Random Number Applications
Random Number Functions
9.1: MATLAB Random Number Functions
range
3.11: Chapter Review
rank
13.3: Matrix Inverse, Rank and Determinant
readability
1.7: Clearing the Workspace, Readability Tips
readable
1.3: A Glorified Calculator
reading
8.3: Debugging in Four Acts
real
2.9: Complex Numbers
rectangle
2.3: Examples of assignment scripts
5.8: Matlab Functions with No Outputs
16.4.1: struct Examples and Exercises
recursive
4.9: Recursive for Loops
Relational Operators
6.1: Relational Operators
6.1.1: Relational Operators List
Relative error
3.3: Absolute and Relative Error
3.11: Chapter Review
resistance
10.3: Linear and Polynomial Curve Data Fitting
Retreating
8.3: Debugging in Four Acts
return
7.8: 'while' Loops
Rocket Height
5.5.1: Multiple Inputs Examples and Exercises
root
11.2: fzero() Examples and Exercises
rotate plots
12.8: Cross Product and Torque
Rotation matrices
13.2: Matrix Multiplication Applications
Rotation Matrix
13.1: Basic Matrix Multiplication
round
1.3.1: log(), round(), floor(), and ceil() functions
row
4.11: Common Vector Functions
12.4: Matrix Indexing
Runinating
8.3: Debugging in Four Acts
Running
8.3: Debugging in Four Acts
runtime error
3.2: Bug Taxonomy

S
Scaffolding
3.11: Chapter Review
Scalar division ./
12.5: Matrix Operations, Introduction
Scalar exponent .^
12.5: Matrix Operations, Introduction
Scalar multiplication .*
12.5: Matrix Operations, Introduction
scatter
14: Advanced Plotting
14.1: scatter, scatter3
scatter3
14.1: scatter, scatter3
scientific notation
2.7: Scientific Notation for Large and Small Numbers
script
2.3: Examples of assignment scripts
2.4: Why Scripts?
2.11: Script Documentation
2.15: Chapter Review
5.1: Name Collisions
script, test
2.3: Examples of assignment scripts
seamount
14.1: scatter, scatter3
search
7.5: Working with Conditions
search path
2.4: Why Scripts?
2.15: Chapter Review
semilogx
14.4: Log plots
semilogy
14.4: Log plots

sequence
2.5: The Fibonacci Sequence
3.11: Chapter Review
4.6: Vectors and Sequences
sequences
3.6: Sequences
3.6.1: Sequences Additional Material
series
3.7: Series
3.7.1: Series Additional Material
3.8: Generalization
3.11: Chapter Review
4.9: Recursive for Loops
shop hours
7.9: 'switch' Logic
Siemers
10.3: Linear and Polynomial Curve Data Fitting
Simulink
1.1: Obtaining MATLAB and Octave
sin
1.3.2: Trigonometry Functions
4.11: Common Vector Functions
sind
1.3.2: Trigonometry Functions
4.11: Common Vector Functions
single precision
16.1: Numeric data types
singular
13.3: Matrix Inverse, Rank and Determinant
size
12.3: Common Matrix Functions
Small Numbers
2.7: Scientific Notation for Large and Small Numbers
sort
9.2: MATLAB Statistics Functions
space
5.4.1: Function Names and MATLAB Keywords
special characters
MATLAB Functions, Commands and Special Characters
Spectral Analysis
11.7: Fourier Transforms
sphere
14.1: scatter, scatter3
spline
10.2: Nonlinear Interpolation
sprintf
15.1: sprintf() and fprintf(), Introduction
sqrt
4.11: Common Vector Functions
square brackets
4.1: Creating Vectors
4.2: Methods to Create Vectors in MATLAB
5.6: Functions with Multiple Outputs
5.8: Matlab Functions with No Outputs
standard deviation
5.6: Functions with Multiple Outputs
star
5.8: Matlab Functions with No Outputs
stats
5.6: Functions with Multiple Outputs
std
4.11: Common Vector Functions
5.6: Functions with Multiple Outputs
9.2: MATLAB Statistics Functions
string array
16.5: cell array
struct
16.4: struct Data Type
16.4.1: struct Examples and Exercises
subfunction
11.4: Subfunctions
19.5.1: ODE45 Examples and Exercises
subplot
14.5: Subplots
sum
4.9: Recursive for Loops
4.11: Common Vector Functions
12.3: Common Matrix Functions
sum_squares
5.5: Functions With Multiple Input Variables
summation
3.7.1: Series Additional Material
Sums
8.1: Functions and Vectors
surf
14: Advanced Plotting
14.2: Plot 3D Surface and Contour Plots
surface plot
14: Advanced Plotting
swap
2.17: Exercises
switch
7.9: 'switch' Logic
symbol
5.4.1: Function Names and MATLAB Keywords
Symbolic Math Toolbox
20: Symbolic Processing
syntax error
3.2: Bug Taxonomy
system of equations
13.5: Systems of Equations Examples and Exercises

T
table data type
16.3: table Data Type
tan
4.11: Common Vector Functions
tand
4.11: Common Vector Functions
tank
18.1.1: Problem Set
target
2.15: Chapter Review
Taylor's series
3.7.1: Series Additional Material
temperature
7.2: Flowcharts
7.3: The input() Function
text
4.9: Recursive for Loops
10.3: Linear and Polynomial Curve Data Fitting
time sequence
3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop
times_two
5.2.1: Function Case Study (Lambert)
title
4.8.1: Plotting Vectors Additional Material
torque
12.8: Cross Product and Torque
total
3.7: Series
3.7.1: Series Additional Material
total_new
3.7.1: Series Additional Material
transpose
4.11: Common Vector Functions
12.1: Creating Matrices and Arrays
12.5: Matrix Operations, Introduction
12.6: Matrix Tables and Nested Loops
trapezoid
18.1: Integration- Computing the Area Under a Curve
trapz
18.1.2: The integral() function
triangle
2.3: Examples of assignment scripts
5.8: Matlab Functions with No Outputs
16.4.1: struct Examples and Exercises
trigonometry
1.3: A Glorified Calculator
True
6.3: if, else, elseif
try
7.1: if, elseif, else with Error Checking
try, catch
7.1: if, elseif, else with Error Checking

U
underscore
5.4.1: Function Names and MATLAB Keywords
user interface
1.4: MATLAB User Interface
1.5: Octave User Interface

V
variable
5.2: Defining Functions
Variable Assignments
1.11: Chapter 1 videos English and Espanol
variable names
1.6.1: More About Valid Variable Names
variable numbers of inputs
5.10: Functions with variable numbers of inputs or outputs
variable updating
3.1: Updating Variables
variables
3.1: Updating Variables
vector
4.1: Creating Vectors
4.2: Methods to Create Vectors in MATLAB
4.6: Vectors and Sequences
4.8.1: Plotting Vectors Additional Material
4.11: Common Vector Functions
vector arithmetic
4.3: Vector Arithmetic
vector elements
4.4: Selecting Elements
vector index
4.4: Selecting Elements
vector operators
4.10: Common Vector Operations

vectorizing Functions
8.1: Functions and Vectors
vectors
8.1.2: User-Defined Functions of Vectors Examples and Exercises
Vectors as Input Variables
8.1: Functions and Vectors
Vectors as Output Variables
8.1: Functions and Vectors
Vehicle
10.3: Linear and Polynomial Curve Data Fitting
verbatim
2.15: Chapter Review
videos
1.11: Chapter 1 videos English and Espanol
15.4: Chapter 15 Videos English and Espanol
videos en espanol
1.1: Obtaining MATLAB and Octave
view
14.1: scatter, scatter3
voltage
4.8.1: Plotting Vectors Additional Material
10.3: Linear and Polynomial Curve Data Fitting

W
wear a coat
7.2: Flowcharts
weather
10.2: Nonlinear Interpolation
while
7.8: 'while' Loops
Windows
1.1: Obtaining MATLAB and Octave
WolframAlpha
20: Symbolic Processing
working directory
1.4: MATLAB User Interface
1.5: Octave User Interface
workspace
5.1: Name Collisions
5.2.1: Function Case Study (Lambert)
5.11: How Functions Work

X
xlabel
4.8.1: Plotting Vectors Additional Material
4.9: Recursive for Loops
19.8: Labeling Axes
xlim
14.1: scatter, scatter3
xlsread
15.3: Reading from an Excel file in MATLAB and Octave
xlswrite
15.3: Reading from an Excel file in MATLAB and Octave
xor operator
7.4: Compound Logical Operators

Y
ylabel
4.8.1: Plotting Vectors Additional Material
4.9: Recursive for Loops
19.8: Labeling Axes
ylim
14.1: scatter, scatter3

Z
zero finding
11: Zero Finding and Fourier Transforms
zeros
4.2: Methods to Create Vectors in MATLAB
12.1: Creating Matrices and Arrays
zlim
14.1: scatter, scatter3

Glossary

Recursive for loop | A loop in which the calculations for each iteration depend on the results of the previous iteration. It has one or more recursive variables.

Recursive script or function | A script or function that calls itself. This repeats itself many times, until either code logic ends the loop or until the maximum number of recursive calls in MATLAB is reached. Recursive scripts and functions without logic to to end the loop are almost always mistakes. There are a very few specialized functions that use recursive calls to themselves; these must have logic to end the recursion.

Recursive variable | A variable whose updated value is a function of it's previous value.

Detailed Licensing

Overview

Title: Matlab and Octave Programming for STEM Applications (Smith)

Webpages: 241

Applicable Restrictions: Noncommercial

All licenses found:

- CC BY-NC-SA 4.0: 94.2% (227 pages)
- Undeclared: 5.8% (14 pages)

By Page

- Matlab and Octave Programming for STEM Applications (Smith) - *CC BY-NC-SA 4.0*
  - Front Matter - *CC BY-NC-SA 4.0*
    - Front Matter - *Undeclared*
      - TitlePage - *Undeclared*
      - InfoPage - *Undeclared*
      - Table of Contents - *Undeclared*
      - Licensing - *Undeclared*
    - TitlePage - *CC BY-NC-SA 4.0*
    - InfoPage - *CC BY-NC-SA 4.0*
    - Table of Contents - *Undeclared*
    - Licensing - *CC BY-NC-SA 4.0*
    - Preface - *CC BY-NC-SA 4.0*
      - 1.1: Notes to the Instructor - *CC BY-NC-SA 4.0*
    - Authors and Sources - *CC BY-NC-SA 4.0*
    - Purpose of This Book - *CC BY-NC-SA 4.0*
    - How this Book Was Assembled - *CC BY-NC-SA 4.0*
    - Back Matter - *Undeclared*
      - Index - *Undeclared*
      - Glossary - *Undeclared*
      - Detailed Licensing - *Undeclared*
  - 1: Modeling, Simulation, and MATLAB’s Interpreter - *CC BY-NC-SA 4.0*
    - 1.1: Obtaining MATLAB and Octave - *CC BY-NC-SA 4.0*
    - 1.2: Modeling - *CC BY-NC-SA 4.0*
    - 1.3: A Glorified Calculator - *Undeclared*
      - 1.3.1: log(), round(), floor(), and ceil() functions - *CC BY-NC-SA 4.0*
      - 1.3.2: Trigonometry Functions - *CC BY-NC-SA 4.0*
    - 1.4: MATLAB User Interface - *CC BY-NC-SA 4.0*
    - 1.5: Octave User Interface - *CC BY-NC-SA 4.0*
    - 1.6: Variables and Assignment Statements - *CC BY-NC-SA 4.0*
      - 1.6.1: More About Valid Variable Names - *CC BY-NC-SA 4.0*
    - 1.7: Clearing the Workspace, Readability Tips - *CC BY-NC-SA 4.0*
    - 1.8: Errors - *CC BY-NC-SA 4.0*
    - 1.9: Built-in Function Documentation - *CC BY-NC-SA 4.0*
    - 1.10: Chapter Review - *CC BY-NC-SA 4.0*
    - 1.11: Chapter 1 videos English and Espanol - *CC BY-NC-SA 4.0*
    - 1.12: Exercises - *CC BY-NC-SA 4.0*
  - 2: MATLAB/Octave Scripts - *CC BY-NC-SA 4.0*
    - 2.1: How to Create a Script File and Valid File Names - *CC BY-NC-SA 4.0*
    - 2.2: Your First Script - *CC BY-NC-SA 4.0*
      - 2.2.1: .mlx Live Editor Files - *CC BY-NC-SA 4.0*
    - 2.3: Examples of assignment scripts - *CC BY-NC-SA 4.0*
    - 2.4: Why Scripts? - *CC BY-NC-SA 4.0*
    - 2.5: The Fibonacci Sequence - *CC BY-NC-SA 4.0*
    - 2.6: Floating-Point Numbers - *CC BY-NC-SA 4.0*
    - 2.7: Scientific Notation for Large and Small Numbers - *CC BY-NC-SA 4.0*
    - 2.8: Number formats - *CC BY-NC-SA 4.0*
    - 2.9: Complex Numbers - *CC BY-NC-SA 4.0*
    - 2.10: Script Comments - *CC BY-NC-SA 4.0*
    - 2.11: Script Documentation - *CC BY-NC-SA 4.0*
    - 2.12: Assignment and Equality - *CC BY-NC-SA 4.0*
    - 2.13: Understanding Editor Warning and Error messages - *CC BY-NC-SA 4.0*
    - 2.14: Debugging Directory Errors - *CC BY-NC-SA 4.0*
    - 2.15: Chapter Review - *CC BY-NC-SA 4.0*

        - 2.16: Chapter 2 Scripts videos English and Espanol - *CC BY-NC-SA 4.0*
        - 2.17: Exercises - *CC BY-NC-SA 4.0*
            - 2.17.1: More Exercises - *CC BY-NC-SA 4.0*
    - 3: for Loops, Ploting Points, Sequences, and Series - *CC BY-NC-SA 4.0*
        - 3.1: Updating Variables - *CC BY-NC-SA 4.0*
        - 3.2: Bug Taxonomy - *CC BY-NC-SA 4.0*
        - 3.3: Absolute and Relative Error - *CC BY-NC-SA 4.0*
        - 3.4: for Loops - *CC BY-NC-SA 4.0*
            - 3.4.1: for Loop Supplemental Material - *CC BY-NC-SA 4.0*
        - 3.5: Plotting Points - *CC BY-NC-SA 4.0*
            - 3.5.1: Create a 2D x-y Sine-Wave Plot with a for Loop - *CC BY-NC-SA 4.0*
        - 3.6: Sequences - *CC BY-NC-SA 4.0*
            - 3.6.1: Sequences Additional Material - *CC BY-NC-SA 4.0*
        - 3.7: Series - *CC BY-NC-SA 4.0*
            - 3.7.1: Series Additional Material - *CC BY-NC-SA 4.0*
        - 3.8: Generalization - *CC BY-NC-SA 4.0*
        - 3.9: Incremental Development - *CC BY-NC-SA 4.0*
        - 3.10: for Loop Incremental Development Exercise - *CC BY-NC-SA 4.0*
        - 3.11: Chapter Review - *CC BY-NC-SA 4.0*
        - 3.12: Chapter 3 for loop Videos in English and Espanol - *CC BY-NC-SA 4.0*
        - 3.13: Exercises - *CC BY-NC-SA 4.0*
    - 4: Vectors - *CC BY-NC-SA 4.0*
        - 4.1: Creating Vectors - *CC BY-NC-SA 4.0*
        - 4.2: Methods to Create Vectors in MATLAB - *CC BY-NC-SA 4.0*
        - 4.3: Vector Arithmetic - *CC BY-NC-SA 4.0*
        - 4.4: Selecting Elements - *CC BY-NC-SA 4.0*
        - 4.5: Indexing Errors - *CC BY-NC-SA 4.0*
        - 4.6: Vectors and Sequences - *CC BY-NC-SA 4.0*
        - 4.7: disp() and num2str() Functions and Concatenation - *CC BY-NC-SA 4.0*
        - 4.8: Plotting Vectors - *CC BY-NC-SA 4.0*
            - 4.8.1: Plotting Vectors Additional Material - *CC BY-NC-SA 4.0*
        - 4.9: Recursive for Loops - *CC BY-NC-SA 4.0*
        - 4.10: Common Vector Operations - *CC BY-NC-SA 4.0*
        - 4.11: Common Vector Functions - *CC BY-NC-SA 4.0*
        - 4.12: Chapter Review - *CC BY-NC-SA 4.0*
        - 4.13: Chapter 4 Vector Videos In English and Espanol - *CC BY-NC-SA 4.0*
        - 4.14: Exercises - *CC BY-NC-SA 4.0*
    - 5: User-Defined Functions - *CC BY-NC-SA 4.0*
        - 5.1: Name Collisions - *CC BY-NC-SA 4.0*
        - 5.2: Defining Functions - *CC BY-NC-SA 4.0*
            - 5.2.1: Function Case Study (Lambert) - *CC BY-NC-SA 4.0*
        - 5.3: Function Documentation - *CC BY-NC-SA 4.0*
        - 5.4: Naming Functions - *CC BY-NC-SA 4.0*
            - 5.4.1: Function Names and MATLAB Keywords - *CC BY-NC-SA 4.0*
        - 5.5: Functions With Multiple Input Variables - *CC BY-NC-SA 4.0*
            - 5.5.1: Multiple Inputs Examples and Exercises - *CC BY-NC-SA 4.0*
            - 5.5.2: Mutiple Inputs Example- Ideal Gas law (Lambert) - *CC BY-NC-SA 4.0*
        - 5.6: Functions with Multiple Outputs - *CC BY-NC-SA 4.0*
        - 5.7: Matlab Functions with No Inputs - *CC BY-NC-SA 4.0*
        - 5.8: Matlab Functions with No Outputs - *CC BY-NC-SA 4.0*
        - 5.9: Matlab Functions with No Inputs and No Outputs - *CC BY-NC-SA 4.0*
        - 5.10: Functions with variable numbers of inputs or outputs - *CC BY-NC-SA 4.0*
        - 5.11: How Functions Work - *CC BY-NC-SA 4.0*
        - 5.12: Chapter Review - *CC BY-NC-SA 4.0*
        - 5.13: Chapter 5 Function videos English and Espanol - *CC BY-NC-SA 4.0*
        - 5.14: Exercise - *CC BY-NC-SA 4.0*
    - 6: Conditionals Part 1 - *CC BY-NC-SA 4.0*
        - 6.1: Relational Operators - *CC BY-NC-SA 4.0*
            - 6.1.1: Relational Operators List - *CC BY-NC-SA 4.0*
            - 6.1.2: Octave vs. MATLAB--not equal, end, endif, endfunction - *CC BY-NC-SA 4.0*
        - 6.2: if Statement - *CC BY-NC-SA 4.0*
        - 6.3: if, else, elseif - *CC BY-NC-SA 4.0*
        - 6.4: Incremental Development - *CC BY-NC-SA 4.0*
        - 6.5: Logical Functions - *CC BY-NC-SA 4.0*
        - 6.6: Nested for Loops - *CC BY-NC-SA 4.0*
        - 6.7: Putting It Together - *CC BY-NC-SA 4.0*
        - 6.8: Encapsulation and Generalization - *CC BY-NC-SA 4.0*
        - 6.9: Adding a continue Statement - *CC BY-NC-SA 4.0*

        - 6.10: Chapter Review - *CC BY-NC-SA 4.0*
        - 6.11: Chapter 6 if-logic videos English and Espanol - *CC BY-NC-SA 4.0*
        - 6.12: Exercise - *CC BY-NC-SA 4.0*
    - 7: Conditionals Part 2- Series, Switch Logic, Error Checking, and While Loops - *CC BY-NC-SA 4.0*
        - 7.1: if, elseif, else with Error Checking - *CC BY-NC-SA 4.0*
        - 7.2: Flowcharts - *CC BY-NC-SA 4.0*
        - 7.3: The input() Function - *CC BY-NC-SA 4.0*
        - 7.4: Compound Logical Operators - *CC BY-NC-SA 4.0*
            - 7.4.1: Compound if, elseif Logic Examples - *CC BY-NC-SA 4.0*
        - 7.5: Working with Conditions - *CC BY-NC-SA 4.0*
        - 7.6: Shortcut and, or, Logical Operations - *CC BY-NC-SA 4.0*
        - 7.7: for loops with break Logic - *CC BY-NC-SA 4.0*
        - 7.8: 'while' Loops - *CC BY-NC-SA 4.0*
        - 7.9: 'switch' Logic - *CC BY-NC-SA 4.0*
        - 7.10: The find() function - *CC BY-NC-SA 4.0*
        - 7.11: Chapter 7 Conditionals, Part 2, Videos English and Espanol - *CC BY-NC-SA 4.0*
    - 8: User-defined Functions of Vectors - *CC BY-NC-SA 4.0*
        - 8.1: Functions and Vectors - *CC BY-NC-SA 4.0*
            - 8.1.1: Built-in MATLAB Functions of Vectors - *CC BY-NC-SA 4.0*
            - 8.1.2: User-Defined Functions of Vectors Examples and Exercises - *CC BY-NC-SA 4.0*
        - 8.2: Computing with Vectors - *CC BY-NC-SA 4.0*
        - 8.3: Debugging in Four Acts - *CC BY-NC-SA 4.0*
            - 8.3.1: Debugging Theorems all in 1 Place - *CC BY-NC-SA 4.0*
        - 8.4: Chapter Review - *CC BY-NC-SA 4.0*
        - 8.5: Chapter 8 Function Videos English and Espanol - *CC BY-NC-SA 4.0*
    - 9: MATLAB Random Number and Statistics Functions - *CC BY-NC-SA 4.0*
        - 9.1: MATLAB Random Number Functions - *CC BY-NC-SA 4.0*
        - 9.2: MATLAB Statistics Functions - *CC BY-NC-SA 4.0*
        - 9.3: Statistical Plots - *CC BY-NC-SA 4.0*
        - 9.4: Random Number Applications - *CC BY-NC-SA 4.0*
        - 9.5: Chapter 9 Random Number Videos English and Espanol - *CC BY-NC-SA 4.0*
    - 10: Interpolation and Curve Fitting - *CC BY-NC-SA 4.0*
        - 10.1: Linear Interpolation - *CC BY-NC-SA 4.0*
            - 10.1.1: Linear Interpolation Problem Set - *CC BY-NC-SA 4.0*
        - 10.2: Nonlinear Interpolation - *CC BY-NC-SA 4.0*
        - 10.3: Linear and Polynomial Curve Data Fitting - *CC BY-NC-SA 4.0*
        - 10.4: Polynomial Fitting Problem Set - *CC BY-NC-SA 4.0*
        - 10.5: Interactive Fitting Tools - *CC BY-NC-SA 4.0*
        - 10.6: Chapter 10 Curve Fitting Videos in English and Espanol - *CC BY-NC-SA 4.0*
    - 11: Zero Finding and Fourier Transforms - *CC BY-NC-SA 4.0*
        - 11.1: Solving Nonlinear Equations with fzero() - *CC BY-NC-SA 4.0*
        - 11.2: fzero() Examples and Exercises - *CC BY-NC-SA 4.0*
        - 11.3: Using fzero() to find the intersection of 2 functions - *CC BY-NC-SA 4.0*
        - 11.4: Subfunctions - *CC BY-NC-SA 4.0*
        - 11.5: Anonymous (or In-Line) Functions - *CC BY-NC-SA 4.0*
        - 11.6: Debugging- More Name Collisions - *CC BY-NC-SA 4.0*
        - 11.7: Fourier Transforms - *CC BY-NC-SA 4.0*
        - 11.8: Chapter Review - *CC BY-NC-SA 4.0*
        - 11.9: Chapter 11 Zero finding Videos English and Espanol - *CC BY-NC-SA 4.0*
        - 11.10: Exercises - *CC BY-NC-SA 4.0*
    - 12: MATLAB Matrices (Arrays) - *CC BY-NC-SA 4.0*
        - 12.1: Creating Matrices and Arrays - *CC BY-NC-SA 4.0*
        - 12.2: Matrices Overview - *CC BY-NC-SA 4.0*
        - 12.3: Common Matrix Functions - *CC BY-NC-SA 4.0*
        - 12.4: Matrix Indexing - *CC BY-NC-SA 4.0*
        - 12.5: Matrix Operations, Introduction - *CC BY-NC-SA 4.0*
        - 12.6: Matrix Tables and Nested Loops - *CC BY-NC-SA 4.0*
        - 12.7: Dot Product - *CC BY-NC-SA 4.0*
        - 12.8: Cross Product and Torque - *CC BY-NC-SA 4.0*
        - 12.9: Chapter 12 Matrix Videos English and Espanol - *CC BY-NC-SA 4.0*
    - 13: Linear Algebra - *CC BY-NC-SA 4.0*
        - 13.1: Basic Matrix Multiplication - *CC BY-NC-SA 4.0*
        - 13.2: Matrix Multiplication Applications - *CC BY-NC-SA 4.0*

    - 13.3: Matrix Inverse, Rank and Determinant - *CC BY-NC-SA 4.0*
    - 13.4: Solving Systems of Equations - *CC BY-NC-SA 4.0*
    - 13.5: Systems of Equations Examples and Exercises - *CC BY-NC-SA 4.0*
    - 13.6: Over-Determined Systems of Equations - *CC BY-NC-SA 4.0*
    - 13.7: Chapter 13 Linear Algebra Videos in English and Espanol - *CC BY-NC-SA 4.0*
  - 14: Advanced Plotting - *CC BY-NC-SA 4.0*
    - 14.1: scatter, scatter3 - *CC BY-NC-SA 4.0*
    - 14.2: Plot 3D Surface and Contour Plots - *CC BY-NC-SA 4.0*
    - 14.3: Advanced Plot Formatting - *CC BY-NC-SA 4.0*
    - 14.4: Log plots - *CC BY-NC-SA 4.0*
    - 14.5: Subplots - *CC BY-NC-SA 4.0*
    - 14.6: Chapter 14 Advanced Plotting Videos in English and Espanol - *CC BY-NC-SA 4.0*
  - 15: Formatted Input and Output - *Undeclared*
    - 15.1: sprintf() and fprintf(), Introduction - *CC BY-NC-SA 4.0*
    - 15.2: fprintf() to a file - *CC BY-NC-SA 4.0*
    - 15.3: Reading from an Excel file in MATLAB and Octave - *CC BY-NC-SA 4.0*
    - 15.4: Chapter 15 Videos English and Espanol - *CC BY-NC-SA 4.0*
  - 16: MATLAB Structures and Data Types - *CC BY-NC-SA 4.0*
    - 16.1: Numeric data types - *CC BY-NC-SA 4.0*
    - 16.2: Comparing Character Strings - *CC BY-NC-SA 4.0*
    - 16.3: table Data Type - *CC BY-NC-SA 4.0*
    - 16.4: struct Data Type - *CC BY-NC-SA 4.0*
      - 16.4.1: struct Examples and Exercises - *CC BY-NC-SA 4.0*
    - 16.5: cell array - *CC BY-NC-SA 4.0*
    - 16.6: Chapter 16 Data Types Videos in English and Espanol - *CC BY-NC-SA 4.0*
  - 17: Top-Down Design, Agile Software Development, and Object-Oriented Programming in MATLAB - *CC BY-NC-SA 4.0*
    - 17.1: Top-Down Design - *CC BY-NC-SA 4.0*
    - 17.2: Agile Software Development - *CC BY-NC-SA 4.0*
    - 17.3: Object-Oriented Programming, Part 1 - *CC BY-NC-SA 4.0*
    - 17.4: Object-Oriented Programming, Part 2 - *CC BY-NC-SA 4.0*
    - 17.5: Chapter 17 Videos in English and Espanol - *CC BY-NC-SA 4.0*
  - 18: Integration and Differentiation - *CC BY-NC-SA 4.0*
    - 18.1: Integration- Computing the Area Under a Curve - *CC BY-NC-SA 4.0*
      - 18.1.1: Problem Set - *CC BY-NC-SA 4.0*
      - 18.1.2: The integral() function - *Undeclared*
    - 18.2: Numerical Differentiation - *CC BY-NC-SA 4.0*
    - 18.3: Chapter 18 Differentiation and Integration Videos in English and Espanol - *CC BY-NC-SA 4.0*
  - 19: Ordinary Differential Equations - *CC BY-NC-SA 4.0*
    - 19.1: Functions and Files - *CC BY-NC-SA 4.0*
    - 19.2: Differential Equations - *CC BY-NC-SA 4.0*
    - 19.3: Euler’s Method - *CC BY-NC-SA 4.0*
    - 19.4: Implementing Euler’s Method - *CC BY-NC-SA 4.0*
    - 19.5: Solving ODEs with ode45 - *CC BY-NC-SA 4.0*
      - 19.5.1: ODE45 Examples and Exercises - *CC BY-NC-SA 4.0*
    - 19.6: Time Dependence - *CC BY-NC-SA 4.0*
    - 19.7: What Could Go Wrong? - *CC BY-NC-SA 4.0*
    - 19.8: Labeling Axes - *CC BY-NC-SA 4.0*
    - 19.9: Chapter Review - *CC BY-NC-SA 4.0*
    - 19.10: Chapter 19 ODE Videos in English and Espanol - *CC BY-NC-SA 4.0*
    - 19.11: Exercise - *CC BY-NC-SA 4.0*
  - 20: Symbolic Processing - *CC BY-NC-SA 4.0*
  - Back Matter - *CC BY-NC-SA 4.0*
    - Detailed Licensing - *CC BY-NC-SA 4.0*
    - MATLAB Functions, Commands and Special Characters - *CC BY-NC-SA 4.0*
    - Index - *CC BY-NC-SA 4.0*
    - Glossary - *CC BY-NC-SA 4.0*
    - Detailed Licensing - *Undeclared*